Developing and Testing Pervasive Computing Applications: A Tool-Based Methodology

Julien Bruneau

UNIVERSITÉ DE BORDEAUX

Département de formation doctorale en informatique

École doctorale EDMI Bordeaux
N° d'ordre : 4518

Développement et Test d'Applications d'Informatique Ubiquitaire : Une Méthodologie Outillée

THÈSE

soutenue le 16 Mai 2012

pour l'obtention du

Doctorat de l'Université de Bordeaux 1

(spécialité informatique)

par

Julien Bruneau

Jury

Président :	Marc Phalippou,	Directeur de l'Enseirb-Matmeca
Rapporteurs :	Michel Banâtre,	Directeur de Recherche à Inria Rennes
	Julie A McCann,	Professeur à l'Imperial College de Londres
Examinateurs :	Emilie Balland,	Chargée de recherche à Inria Bordeaux
	Charles Consel,	Professeur à l'Institut Polytechnique de Bordeaux
	Walid Taha,	Professeur à l'Université d'Halmstad

ABSTRACT

DEVELOPING AND TESTING PERVASIVE COMPUTING APPLICATIONS: A TOOL-BASED METHODOLOGY

Despite much progress, developing a pervasive computing application remains a challenge because of a lack of conceptual frameworks and supporting tools. This challenge involves coping with heterogeneous devices, overcoming the intricacies of distributed systems technologies, working out an architecture for the application, and encoding it into a program. Moreover, testing pervasive computing applications is problematic because it requires acquiring, testing and interfacing a variety of software and hardware entities. This process can rapidly become costly and time-consuming when the target environment involves many entities.

This thesis proposes a tool-based methodology for developing and testing pervasive computing applications. Our methodology first provides the DiaSpec design language that allows to define a taxonomy of area-specific building-blocks, abstracting over their heterogeneity. This language also includes a layer to define the architecture of an application. Our tool suite includes a compiler that takes DiaSpec design artifacts as input and generates a programming framework that supports the implementation and testing stages.

To address the testing phase, we propose an approach and a tool integrated in our tool-based methodology, namely DiaSim. Our approach uses the testing support generated by DiaSpec to transparently test applications in a simulated physical environment. The simulation of an application is rendered graphically in a 2D visualization tool.

We combined DiaSim with a domain-specific language for describing physical environment phenomena as differential equations, allowing a physically-accurate testing. DiaSim has been used to simulate various pervasive computing systems in different application areas. Our simulation approach has also been applied to an avionics system, which demonstrates the generality of our parameterized simulation approach.

KEYWORDS: Software Architecture, Domain-Specific Language, Generative Programming, Testing, Simulation

RÉSUMÉ

DÉVELOPPER ET TESTER DES APPLICATIONS D'INFORMATIQUE UBIQUITAIRE : UNE MÉTHODOLOGIE OUTILLÉE

Malgré des progrès récents, développer une application d'informatique ubiquitaire reste un défi à cause d'un manque de canevas conceptuels et d'outils aidant au développement. Ce défi implique de prendre en charge des objets communicants hétérogènes, de surmonter la complexité des technologies de systèmes distribués, de définir l'architecture d'une application, et d'encoder cela dans un programme. De plus, tester des applications d'informatique ubiquitaire est problématique car cela implique d'acquérir, de tester et d'interfacer une variété d'entités logicielles et matérielles. Ce procédé peut rapidement devenir coûteux en argent et en temps lorsque l'environnement ciblé implique de nombreuses entités.

Cette thèse propose une méthodologie outillée pour développer et tester des applications d'informatique ubiquitaire. Notre méthodologie fournit tout d'abord le langage de conception DiaSpec. Ce langage permet de définir une taxonomie d'entités spécifiques à un domaine applicatif, s'abstrayant ainsi de leur hétérogénéité. Ce langage inclut également une couche permettant de définir l'architecture d'une application. Notre suite outillée fournit un compilateur qui, à partir de descriptions DiaSpec, génère un canevas de programmation guidant les phases d'implémentation et de test.

Afin d'aider à la phase de test, nous proposons une approche de simulation et un outil intégré dans notre méthodologie outillée : l'outil DiaSim. Notre approche utilise le support de test généré par DiaSpec pour tester les applications de manière transparente dans un environnement physique simulé. La simulation d'une application est rendue graphiquement dans un outil de visualisation 2D.

Nous avons combiné DiaSim avec un langage dédié permettant de décrire les phénomènes physiques en tant qu'équations différentielles, permettant des simulations réalistes. DiaSim a été utilisé pour simuler des applications dans des domaines applicatifs variés. Notre approche de simulation a également été appliquée à un système avionique, démontrant la généralité de notre approche de simulation.

MOTS CLÉS : Architecture Logicielle, Langage Dédié, Programmation Générative, Test, Simulation

PUBLICATIONS

The works discussed in this thesis have been previously published.

JOURNALS

- "DiaSim: A Simulator for Pervasive Computing Applications", in *Software: Practice and Experience*, 2012, Julien Bruneau and Charles Consel

- "Towards a Tool-based Development Methodology for Pervasive Computing Applications," in *IEEE Transactions on Software Engineering*, 2011, Damien Cassou, Julien Bruneau, Charles Consel, and Emilie Balland

- "DiaSuite: a Tool Suite To Develop Sense/Compute/Control Applications", in *Science of Computer Programming*, April 2012, Benjamin Bertran, Julien Bruneau, Damien Cassou, Nicolas Loriant, Emilie Balland, and Charles Consel

INTERNATIONAL CONFERENCES

- "DiaSim: A Parameterized Simulator for Pervasive Computing Applications," in *Mobiquitous'09: Proceedings of the 6th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, 2009, Julien Bruneau, Wilfried Jouve, and Charles Consel

- "Virtual Testing for Smart Buildings," in *IE'12: Proceedings of the The 8th International Conference on Intelligent Environments*, 2012, Julien Bruneau, Charles Consel, Marcia O'Malley, Walid Taha, and Wail Masry Hannourah

DEMONSTRATIONS

- "A Tool Suite to Prototype Pervasive Computing Applications," in *PerCom'10: Proceedings of the 8th International Conference on Pervasive Computing and Communications*, 2010, Damien Cassou, Julien Bruneau et Charles Consel

– "A Parameterized Simulator for Pervasive Computing Applications," in *ICPS'09: Proceedings of the ACM International Conference on Pervasive Services*, 2009, Julien Bruneau, Alexandre Blanquart, Nicolas Loriant, and Charles Consel
Best Demo Award

– "DiaSim: A Parameterized Simulator for Pervasive Computing Applications," in *PerCom'09: Proceedings of the 7th International Conference on Pervasive Computing and Communications*, 2009, Wilfried Jouve, Julien Bruneau, and Charles Consel

POSTERS

– "Preliminary Results in Virtual Testing for Smart Buildings," in *Mobiquitous'10: Proceedings of the 7th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, 2010, Julien Bruneau, Charles Consel, Marcia O'Malley, Walid Taha, and Wail Masry Hannourah

– "Towards a Tool-based Development Methodology for Sense/-Compute/Control Applications," in *SPLASH'10: Proceedings of the 1st International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, 2010, Damien Cassou, Julien Bruneau, Julien Mercadal, Quentin Enard, Emilie Balland, Nicolas Loriant, and Charles Consel

TECHNICAL REPORTS

– "Design-driven Development of Safety-Critical Applications: A Case Study in Avionics", Julien Bruneau, Quentin Enard, Stéphanie Gatti, Emilie Balland, and Charles Consel

REMERCIEMENTS

Cette thèse n'aurait pas pu se dérouler dans de si bonnes conditions sans l'aide de nombreuses personnes.

Je tiens tout d'abord à remercier chaleureusement mon directeur de thèse Charles Consel pour m'avoir guidé, encouragé, conseillé et fait beaucoup voyager pendant ces presque quatre années de thèse. Son haut niveau d'exigence et son perfectionnisme m'ont obligé à donner le meilleur de moi même tout au long de ma thèse.

Je remercie également chaleureusement Emilie Balland pour ses nombreux conseils et son aide précieuse dans toutes les activités d'une thèse en informatique, de l'implémentation jusqu'à l'écriture de papier, en passant par les présentations. Je la remercie également d'avoir accepté de faire partie de mon jury de thèse.

Mes remerciements vont également à Walid Taha pour toute son aide lors de nos travaux collaboratifs, pour sa gentillesse, pour l'hospitalité dont il a fait preuve envers moi lors de mon séjour à Rice University, et aussi pour m'avoir fait l'honneur de participer à mon jury de thèse.

Je remercie Julie McCann, professeur à l'Imperial College de Londres, et Michel Banatre, directeur de recherche Inria, de m'avoir fait l'honneur d'accepter d'être rapporteur de ma thèse. Mes remerciements vont également à Marc Phalippou, directeur de l'école d'ingénieur ENSEIRB-Matmeca, d'avoir accepté de présider mon jury de thèse.

Je remercie très chaleureusement tous les membres de l'équipe Phoenix, avec qui cela a été un plaisir de travailler pendant ces quatre années et sans qui ma thèse ne serait pas ce qu'elle est. Je voudrais donc remercier Benjamin, parce que son sens critique et nos très nombreuses discussions sur les aspects conceptuels, techniques et musicaux ont été d'une très grande aide tout au long de ma thèse, Zoé pour nos discussions et ses conseils précieux en tant qu'utilisatrice de DiaSim, pour avoir tenté de m'expliquer la sémantique dénotationnelle, et pour sa bonne humeur communicative, Nicolas pour toutes ses idées d'amélioration de DiaSim que je n'aurais finalement pas implémentées et pour ses nombreux points "le saviez-vous ?" qui ont grandement élargis le spectre de mes connaissances, Quentin pour toute son aide dans l'implémentation de l'auto-pilote, pour toutes nos discussions sur des sujets variés et parce que ça a été un plaisir de partager le même bureau, Wilfried pour m'avoir guidé au début de ma thèse et pour tous ces matchs intenses au baby foot et au ten-

nis, Damien C. pour son soutien lors de l'écriture de papiers et parce qu'il m'a fait découvrir le Humble Bundle, Julien pour ses conseils et son sens critique qui m'ont poussé à faire de mon mieux, Henner pour toutes ces parties de baby foot endiablées, Hong pour toute son aide dans les démarches administratives à la fin de ma thèse, Pengfei pour toutes nos discussions et pour nos parties de Starcraft qui m'ont permis de me détendre dans la période stressante de fin de thèse, Stéphanie pour son expertise et son soutien lors de nos rencontres avec Thales, Alexandre pour sa bonne humeur nordique apportée au quotidien au labo. Enfin je remercie Amélie, Damien M., Luc et Charles Jr. qui sont arrivés dans l'équipe à la fin de thèse et avec qui j'ai passé de très bons moments au labo ou en dehors. Je remercie Sylvie et Chrystel pour leur aide au quotidien dans toutes les démarches administratives.

Enfin, je remercie mes parents, ma soeur Chloé et mes grands-parents pour être venus assister à ma soutenance et pour m'avoir soutenu tout au long de ma thèse. Mes derniers remerciements vont à Julie, pour son soutien sans faille pendant les moments stressants de ma thèse.

CONTENTS

I CONTEXT 1

1 INTRODUCTION 3
- 1.1 Requirements 4
- 1.2 Contributions 6
- 1.3 Outline 7

2 OVERVIEW 9
- 2.1 Case Study: a Heating Control System 9
- 2.2 Sense-Compute-Control Paradigm 10
- 2.3 A Design-driven Methodology 11
- 2.4 A Tool-based Methodology 12
- 2.5 Testing Pervasive Computing Applications 14

II DEVELOPING PERVASIVE COMPUTING APPLICATIONS 17

3 DESIGNING AN APPLICATION 19
- 3.1 Designing the Taxonomy 19
- 3.2 Architecting the Application 21

4 IMPLEMENTING AN APPLICATION 25
- 4.1 Programming Framework Generator 25
- 4.2 Generated Programming Framework 25
- 4.3 Implementation of Entities 26
- 4.4 Developing the Application Logic 28
- 4.5 Entity Discovery 31

III TESTING PERVASIVE COMPUTING APPLICATIONS 33

5 TESTING AN APPLICATION WITH DIASIM 35
- 5.1 Simulation Model 35
- 5.2 Developing a Simulation 38
- 5.3 Testing Support 44

6 DIASIM IMPLEMENTATION 47
- 6.1 Implementation 47
- 6.2 Applications 50

7 DIASIM VALIDATION 53
- 7.1 DiaSim Evaluation 53
- 7.2 Discussion 56

8 PHYSICALLY-ACCURATE TESTING 59
- 8.1 Modeling the Physical Environment 60
- 8.2 Mapping the Models into Executable Simulation Codes 63
- 8.3 Monitoring Physically-Accurate Simulation 64
- 8.4 A Virtual Experiment 65
- 8.5 Discussion 69

9 GENERALIZATION OF OUR SIMULATION APPROACH 71
9.1 An Integrated Approach to Simulation 71
9.2 Realistic Simulation of an Avionics System 72

IV CONCLUSION 77

10 RELATED WORK 79
10.1 Model-Driven Engineering 79
10.2 Architecture Description Languages 80
10.3 Context management middlewares 80
10.4 Programming Frameworks 81
10.5 Simulators 82

11 CONCLUSION 85
11.1 Assessments 86
11.2 Ongoing and future work 87

BIBLIOGRAPHY 91

LIST OF FIGURES

Figure 1 The SCC paradigm. 10

Figure 2 Flowchart of the development activities of our tool-based methodology. Multiple inputs for an activity require synchronization; multiple outputs enable parallelization. 11

Figure 3 Development support provided by the DiaSuite tools. 12

Figure 4 Extract of the heating control system taxonomy. DiaSpec keywords are printed in **bold**. 20

Figure 5 Specification of the heating control system. 22

Figure 6 Architecture of the heating control system. DiaSpec keywords are printed in **bold**. 23

Figure 7 Structure of the DiaGen compiler. 25

Figure 8 The Java abstract class `AbstractMotionDetector` generated by DiaGen from the declaration of the MotionDetector entity (Figure 4, lines 5 to 7). 27

Figure 9 The Java abstract class `AbstractHeater` generated by DiaGen from the declaration of the Heater entity (Figure 4, lines 13 to 15). 27

Figure 10 A developer-supplied Java implementation of a `MotionDetector` entity. This class extends the generated abstract class shown in Figure 8. The implementation relies on a third party library: `motionDetected` is a callback method from the `MotionDetectionListener` interface. 29

Figure 11 A developer-supplied implementation of the `AverageTemperature` context. 30

Figure 12 Implementation of the `HeatRegulator` controller. 30

Figure 13 Simulation model. 38

Figure 14 Correspondence between real and simulation programming frameworks. 39

Figure 15 Implementation of the generated `SimulatedMotionDetector` class. 39

Figure 16 Implementation of a simulated `MotionDetector` entity 40

Figure 17 Implementation of the `MyAgentModel` class used in the heating control system simulation. This class is responsible for publishing motion detection events when simulated people come within a range of a motion detector. 41
Figure 18 Extract of the `studentagenda.xml` XML file. 43
Figure 19 Class diagram of the implementation of the heat regulator simulation. 43
Figure 20 DiaSim architecture. 45
Figure 21 DiaSim scenario editor. The DiaSim editor is parameterized by an entity taxonomy. The entities defined in the taxonomy are displayed on the left panel of the graphical user interface. The entities can be dragged and dropped on the central panel to add simulated entity instances into the simulated environment. 48
Figure 22 DiaSim simulation renderer. The simulated environment is displayed in the left part of the graphical user interface. The red popups transparently displayed above the simulated entities indicate that the entity has realized an interaction. More information about the simulated people and simulated entities can be found on the right of the graphical user interface. 49
Figure 23 This graph represents the average CPU usage with respect to the simulation speed. The CPU usage has been evaluated during a period of low activity for the simulated agents and during a period of high activity. 55
Figure 24 Temperature specification in Acumen. 63
Figure 25 2D graphical rendering of a virtual house. 65
Figure 26 Comparison of different algorithms for regulating the temperature in terms of energy use. 67
Figure 27 Comparison of the comfort provided by global and local temperature management in the house. 68
Figure 28 Comparison of the time of heating required by global and local temperature management in the house. 68

Figure 29 Specification of the flight guidance application application. 73
Figure 30 Simulation model of an avionics application. 75
Figure 31 Extract of the implementation of a simulated `InertialUnit`. 75
Figure 32 Screenshot of a simulated flight. 76

LIST OF TABLES

Table 1 Components of the heating control system. 23

Table 2 Results of a lab involving 60 Master's level students. 54

Table 3 Distribution of the threads executed during the ENSEIRB simulation. 56

Part I

CONTEXT

1 INTRODUCTION

Pervasive computing applications are being deployed in a growing number of areas, including building automation, assisted living, and supply chain management. Numerous pervasive computing applications coordinate a variety of networked entities collecting data from sensors and reacting by triggering actuators. These entities are either software or hardware. To collect data, sensors process stimuli that are observable changes of the environment (*e.g.*, fire and motion). Triggering actuators is assumed to change the state of the environment.

Besides requiring expertise on underlying technologies, developing a pervasive computing application also involves domain-specific architectural knowledge to collect information relevant for the application, process it, and perform actions. Moreover, such an application needs to implement strategies to manage a variety of scenarios (*e.g.*, fire situations, intrusions, and crowd emergency-escape plans). Consequently, in addition to the challenges of developing any software system, a pervasive computing system needs to validate the environment entities both individually and globally, to identify potential conflicts. For example, a fire manager and an entrance manager could issue contradicting commands to a building door to respectively enable evacuation and ensure security. In practice, the many parameters to take into account for the development of a pervasive computing application can considerably lengthen this process. Not only does this situation have an impact on the application code, but it also involves changes to the physical layout of the target environment, making each iteration time-consuming and error-prone.

Various middlewares and programming frameworks have been proposed to ease the development of pervasive computing applications [21, 34, 58]. However, they require a fully-equipped pervasive computing environment for an application to be run and tested. As a result, an iteration process is still needed, involving the physical setting of the target environment and the application code.

In fact, the development of a pervasive computing system is very similar to the development of an embedded system. Like a pervasive computing system, an embedded system coordinates a number of heterogeneous hardware components that can be viewed as sensors (*e.g.*, GPS and accelerometer) and actuators

(*e.g.,* displays and speakers). Some embedded systems are capable of discovering components dynamically, such as a smartphone detecting bluetooth components. As in the pervasive computing domain, embedded systems developers need to anticipate as wide a range of usage scenarios as possible to program their support. Despite similarities, the embedded systems domain differs from the pervasive computing domain in that it provides approaches and tools to facilitate software development for a system under design. Indeed, embedded systems applications can be developed using programming frameworks, and tested and debugged using emulators [3, 24, 33]. Hardware components are *simulated* via software components that faithfully duplicate their observable behavior. And, the embedded systems application is *emulated*, executing as if it relied on hardware components, without requiring any code change.

1.1 REQUIREMENTS

The study of embedded systems emulators gives us a practical basis for identifying the requirements for pervasive computing systems. We now examine these requirements.

COVERING THE APPLICATION DEVELOPMENT CYCLE Embedded systems tools often cover the whole development cycle, guiding and supporting the developer in each stage of the development. For instance, the Android SDK [33] enables to design an application in a manifest. It provides a Java programming framework to support the development of Android applications. Finally it provides an emulator to test and debug the developed applications. For developing pervasive computing applications, existing general-purpose design frameworks are generic and do not fully support the whole development. To cover this development cycle, a design framework specific to the pervasive computing domain is needed. This domain-specific design framework would improve productivity and facilitate evolution. To make this design framework effective, the conformance between the specification and the implementation must be guaranteed [68, Chap. 9]. Finally, during the application development, tools should enable a comprehensive testing of the application.

ABSTRACTING OVER HETEROGENEITY Embedded systems are required to coordinate heterogeneous devices (*e.g.,* Android systems can coordinate Bluetooth and Wifi devices). Likewise, a pervasive computing application interacts with *entities* (*e.g.,* webcams and calendars), whose heterogeneity tends to percolate

in the application code, cluttering it with low-level details. This situation requires to raise the level of abstraction at which entities are invoked, to factor entity variations out of the application code, and to preserve it from distributed systems dependencies and communication protocol intricacies.

LEVERAGING AREA-SPECIFIC KNOWLEDGE Like embedded systems, pervasive computing systems target a variety of application areas, including home automation, building surveillance and assisted living. Each area corresponds to specific pervasive computing environments, consisting of a taxonomy of entities dedicated to a given activity (*e.g.*, cameras, motion detectors and alarms in the building surveillance area). Thus, knowledge about the entities of each area needs to be shared and made reusable because applications in a given area often share the same classes of entities. Correspondingly, the related stimuli drastically vary with respect to the target area. As a consequence, a simulation tool for the pervasive computing domain is required to deal with different application areas, enabling new types of entities and stimuli to be introduced easily.

TRANSPARENT TESTING A key feature of most embedded systems emulators is that they emulate the execution of an application without requiring any change in the application code. As a result, when the testing phase is completed, the application code can be uploaded as is and its logic does not require further debugging. The same functionality should be provided by a testing tool for pervasive computing applications.

TESTING A WIDE RANGE OF SCENARIOS Some pervasive computing applications address scenarios that cannot be tested because of the nature of stimuli involved (*e.g.*, fire and smoke). In other situations, the scenarios to be tested are large scale in terms of stimuli, entities and physical space they involve. These situations would benefit from a simulation phase to refine the requirements on the constituent entities of the environment, before acquiring them. Regardless of the nature of the target pervasive computing system, its application logic is best tested on a wide range of scenarios, while the system is under design. This strategy allows improvements to be made as early as possible in both its architecture and logic.

SIMULATION RENDERER Like an embedded systems simulator, one for pervasive computing systems needs to simulate and visualize the physical environment. This simulation renderer needs to take into account various features of the pervasive

computing domain. Specifically, it should support visual representations for an open-ended set of entities and stimuli, visual support for scenario monitoring, and debugging facilities to navigate in scenarios in terms of time and space. Some existing approaches propose to visualize the simulation of pervasive computing applications [4, 51, 53]. However, these approaches are limited because they require significant programming effort to address new pervasive computing areas. Furthermore, they do not provide a setting to test applications deterministically. The Lancaster simulator addresses this issue but does not support scenario definition [50]. The PiCSE simulator provides a comprehensive simulation model and generic libraries to create new scenarios. However, users have to manually specialize the simulator for every new application area [59].

1.2 CONTRIBUTIONS

This thesis proposes an approach that covers the development and testing of a pervasive computing application. It takes the form of a tool-based methodology. The main contributions of this thesis are :

A DESIGN-DRIVEN METHODOLOGY We introduce DiaSpec, a design language dedicated to describing both a taxonomy of area-specific entities and pervasive computing application architectures. This design language provides a conceptual framework to support the development of a pervasive computing application. The design is used to provide a dedicated programming support to the developer for the subsequent stages of the development cycle, namely the implementation and testing stages.

A TOOL-BASED METHODOLOGY We have built DiaSuite, a suite of tools which, combined with our design language, provides support for each phase of the development of a pervasive computing application, namely, design, implementation, and testing. DiaSuite relies on a compiler that generates a programming framework from descriptions written in the DiaSpec design language.

A SIMULATION APPROACH To address the testing stage, we propose an approach and a tool integrated in our tool-based methodology, namely DiaSim. DiaSim enables a transparent testing of pervasive computing applications in a simulated physical environment. It also allows the testing of applications in a hybrid environment, combining real and simulated entities. Fi-

nally, it provides a 2D graphical simulation renderer for visually monitoring and debugging pervasive computing applications.

A PHYSICALLY-ACCURATE SIMULATION We have combined DiaSim to Acumen [76], a domain-specific language (DSL) with specialized support for describing continuous systems. The physical characteristics of a physical environment (*e.g.,* temperature or luminosity) are described and simulated using Acumen. This allows us to test applications with DiaSim in a physically-accurate environment.

VALIDATION OF THE SIMULATION APPROACH The generality of our approach has been demonstrated by simulating applications in a variety of pervasive computing areas. The practicality of DiaSim has been shown on a large-scale simulation of an engineering school [38]. Finally, we have also evaluated DiaSim in terms of scalability, performance and usability.

1.3 OUTLINE

The remainder of this dissertation is organized as follows :

- Chapter 2 introduces an overview of the approach presented in this thesis. We first present an example of pervasive computing application: a heating control system. This example is used throughout this dissertation to illustrate our approach. We then introduce the development paradigm underlying our development methodology, namely the Sense-Compute-Control paradigm. Finally, we present an overview of our tool-based methodology for developing and testing pervasive computing applications.
- Chapter 3 presents the DiaSpec language used for designing pervasive computing applications. DiaSpec allows to design a taxonomy of entities from a pervasive computing area. It also enables to design the architecture of pervasive computing applications.
- Chapter 4 introduces the programming framework generated by the DiaSpec compiler from the application design. This support is then used to develop the entities and the application components defined in a DiaSpec design.
- Chapter 5 presents our approach for testing pervasive computing applications. The underlying simulation model and the simulation programming framework provided by our approach are presented in this chapter.
- In Chapter 6, we first present the implementation of our simulation approach, namely DiaSim. DiaSim is composed

of two components: a scenario editor and a simulation renderer. Then, we present a large-scale case study we tested with DiaSim.

- DiaSim is validated in Chapter 7. We first discuss the existing works related to DiaSim. Then, DiaSim is evaluated with respect to scalability, usability, and performance. Finally, we discuss pragmatic issues involved in developing and using a simulated environment.
- Chapter 8 presents how a physically-accurate simulation can be realized using DiaSim and the Acumen DSL. Acumen is used to model and simulate the physical characteristics of the simulated environment. A virtual experiment of our heating control system is presented and evaluated in this chapter. This experiment enables to compare the power and comfort efficiency of different heating strategies.
- Chapter 9 generalizes our simulation approach to an application domain different from the pervasive computing domain, namely the *avionics* domain. This chapter provides insights as to how to use our simulation approach for testing any application that uses the Sense-Compute-Control paradigm.
- A thorough discussion of the works related to our methodology is conducted in Chapter 10.
- Finally, Chapter 11 articulates the conclusions of this work. We also discuss several possible directions for future works.

2

OVERVIEW

This chapter presents an overview of our tool-based methodology for developing and testing pervasive computing applications. This methodology has two main characteristics; it is (1) *design-driven* and (2) *tool-based*. We first introduce a simple case study: a *heating control system*. We use this case study throughout this dissertation. Then, we present the Sense-Compute-Control paradigm. This paradigm is used for architecting the applications developed with our methodology. Then, we show why our methodology is design-driven through its flow of development activities. We also provide an overview of each tool of our methodology that supports these development activities. Finally, we introduce an overview of our approach for testing pervasive computing applications in a simulated physical environment.

2.1 CASE STUDY: A HEATING CONTROL SYSTEM

We illustrate our tool-based methodology with a case study that takes place in one of the areas involved in building management applications, namely, the Heating, Ventilating and Air Conditioning (HVAC) area. HVAC systems are responsible for providing thermal comfort and acceptable indoor air quality to the building occupants. These systems are a standard part of mechanical engineering curricula, see for example [7]. HVAC systems regulate multiple physical properties of a building. For example, temperature and humidity must be regulated so that the occupants feel comfortable. As well, carbon dioxide density needs to be regulated for keeping an acceptable indoor air quality. Finally, the amount of airflow introduced to an air-conditioned zone must be controlled to remain pleasant for the occupants. To regulate these multiple physical characteristics, HVAC systems interact with numerous devices. It retrieves information from sensors such as temperature, humidity and carbon dioxide sensors. It also controls heaters, humidifiers and ventilators to provide comfort and acceptable indoor air quality to occupants.

In our case study, we focus on a specific part of a HVAC system: the *heating control system*. This system regulates the temperature in each room of a building depending on the room occupancy. The building room occupancy is scheduled in a calendar, allowing the heating control system to know in advance the occupancy

planning of each room. The first functionality of this system is to heat a room 15 minutes before a scheduled occupancy if the room temperature is too cold. Thus, the room is at a comfortable temperature when the occupants arrive. The system also manages the heating of rooms that are occupied unexpectedly. The heating control system receives motion detection events to detect unscheduled occupancy. Thus, if someone enters an unoccupied room, the heating control system automatically turns on the heaters.

2.2 SENSE-COMPUTE-CONTROL PARADIGM

Our methodology relies on the Sense-Compute-Control (SCC) paradigm [15]. This paradigm originates from the *sense/compute/control* architectural pattern, promoted by Taylor *et al.* [68]. This paradigm applies to applications that interact with an external environment. Such applications are typical of domains such as building automation, robotics, and autonomic computing.

As depicted in Figure 1, the underlying design pattern consists of *context components* fueled by sensing *entities*. These components refine (aggregate and interpret) the information given by the sensors. These refined data are then passed to *controller components* that trigger actions on entities. For example, the heating control system senses the environment to acquire temperature data. Then, the system uses this raw data to regulate the building temperature accordingly. This temperature regulation is achieved by acting on heaters.

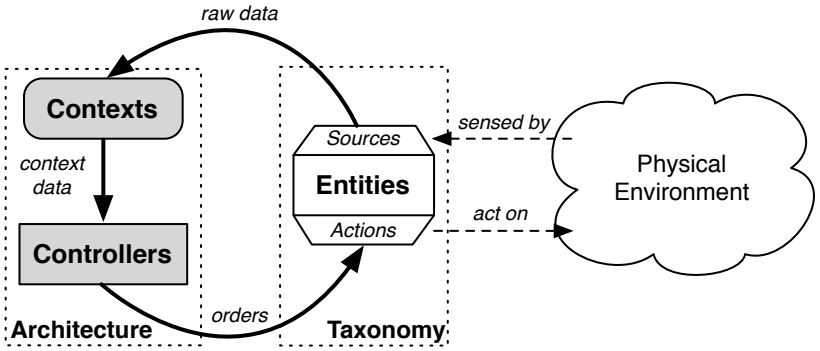

Figure 1: The SCC paradigm.

Like a programming paradigm, the SCC paradigm provides concepts and abstractions to solve a software engineering problem. However, these concepts and abstractions are dedicated to a

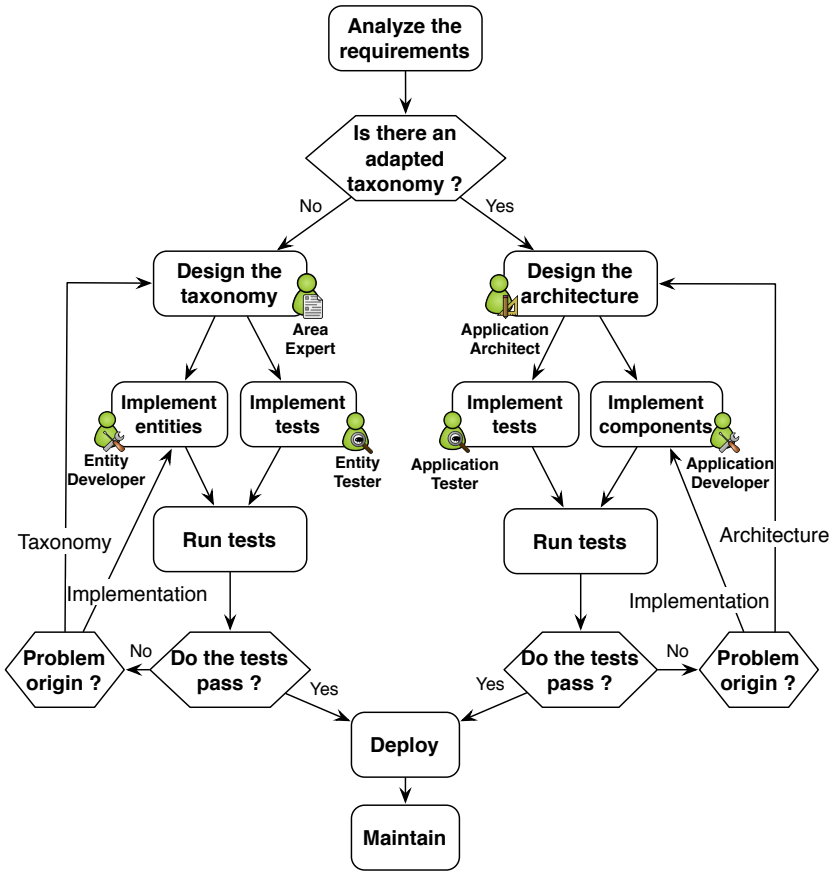

Figure 2: Flowchart of the development activities of our tool-based methodology. Multiple inputs for an activity require synchronization; multiple outputs enable parallelization.

design style, raising the level of abstraction above programming. Because of its dedicated nature, such a development paradigm allows a more disciplined engineering process, as advocated by Shaw [10, 65].

2.3 A DESIGN-DRIVEN METHODOLOGY

An entity is a concept specific to the pervasive computing domain. This concept points out the independence between (1) the development of an entity taxonomy for an area, such as HVAC, and (2) the development of a specific application that orchestrates elements of a taxonomy. This independence leads to two distinct design activities: entity taxonomy design and application design. These design activities, as well as the subsequent implementation and testing activities can be achieved in parallel. Figure 2 outlines the development cycle associated to our methodology and illustrates the independence between these activities. In this figure, a role is associated to each development activity. Even though these

activities are related, they can be achieved by distinct experts in parallel, given that these experts collaborate closely.

In our development methodology, the test implementation can start in parallel with the software component implementation as test implementation only needs information provided by the architecture design and comments provided by the architect. Our approach facilitates test-driven development methodologies (e. g. agile software development [36]) where the test phase strictly precedes the implementation phase. In this way, tests guide the developers of the application.

Along this development life-cycle, our methodology offers tools to assist the experts for each development activity. In particular, the specification is directly used for generating a dedicated programming support.

2.4 A TOOL-BASED METHODOLOGY

Based on this development life-cycle and its identified roles, Figure 3 depicts how our tool suite supports each phase of the proposed methodology:

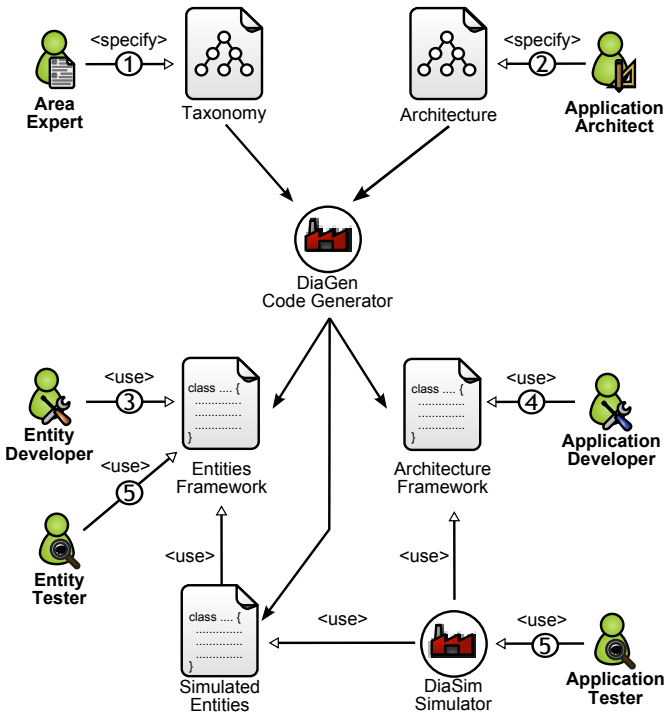

Figure 3: Development support provided by the DiaSuite tools.

DESIGNING THE TAXONOMY Using the taxonomy layer of the DiaSpec language, an expert defines an application area through a catalog of entities, whether hardware or software, that are specific

to a target area (stage ①). A taxonomy allows separation of concerns in that the expert can focus on the high-level description of area-specific entities.

DESIGNING THE ARCHITECTURE Given a taxonomy, the architect can design and structure applications (stage ②). To do so, the DiaSpec language provides an Architecture Description Language (ADL) layer [47] dedicated to describing pervasive computing applications. An architecture description enables the key components of an application to be identified, allowing their implementation to evolve with the requirements (*e.g.*, varying the implementation of the temperature regulation management to optimize energy consumption).

IMPLEMENTING ENTITIES AND COMPONENTS We leverage the taxonomy definition and the architecture description to provide dedicated support to both the entity and the application developers (stages ③ and ④). This support takes the form of a Java programming framework, generated by the DiaGen code generator [14]. The generated programming framework guides the developer with respect to the taxonomy definition and the architecture description. It consists of high-level operations to discover entities and interact with both entities and application components. In doing so, it abstracts away from the underlying distributed technologies, providing further separation of concerns.

TESTING DiaGen generates a simulation support to test pervasive computing applications before their actual deployment (stage ⑤). An application is simulated with DiaSim [11], without requiring any code modification. DiaSim provides a graphical editor to define simulation scenarios and a 2D-renderer to monitor simulated applications. Furthermore, simulated and real entities can be mixed. This hybrid simulation enables an application to migrate incrementally to an actual environment.

DEPLOYING After the testing phase, the system administrator deploys the pervasive computing application. To this end, a distributed systems technology is selected. We have developed a back-end that currently targets the following technologies: Web Services, RMI [14], and SIP [5]. This targeting is transparent for the application code. The variety of these target technologies demonstrates that our development approach separates concerns into well-defined layers. This separation allows to build easily new back-ends if necessary and to smoothly apply them to already existing applications.

MAINTENANCE AND EVOLUTION Our tool-based methodology allows for iterative development of the taxonomy and architecture. This approach allows changes in the taxonomy and architecture during late phases of the cycle.

This dissertation focuses on the design, development and testing stages of the development cycle of pervasive computing applications. For this reason, we will not present the deployment and maintenance stages of our methodology. Further information on the support provided by our methodology for these two stages can be found elsewhere [16].

2.5 TESTING PERVASIVE COMPUTING APPLICATIONS

As in any software engineering domain, testing pervasive computing applications is crucial. However, this domain has specific requirements that prevent generic testing tools from applying to pervasive computing applications [59]. Indeed, pervasive computing applications interact with users and with the physical environment. Generic software testing tools do not cope with neither the simulation of the physical environment, nor the simulation of users in this physical environment.

Coping with these requirements makes the testing of pervasive computing applications challenging. In fact, only a few existing development approaches in the pervasive computing domain address testing: existing development approaches often assume that the system is partially or fully deployed. However, deploying a pervasive computing application for testing purposes can be expensive and time-consuming because it requires to acquire, test, and configure all equipments and software components. Furthermore, some scenarios are difficult to test because they involve exceptional situations such as fire.

To cope with these issues, our methodology provides an approach and a simulator tool for testing pervasive computing applications, named DiaSim. This tool is integrated with our methodology, leveraging declarations provided at earlier development stages. It provides graphical tools for editing simulation scenarios and executing pervasive computing applications against these scenarios in a simulated physical environment. DiaSim leverages the abstraction layer of the generated programming framework for operating entities regardless of their nature (*i.e.*, real or simulated). Thus, pervasive computing applications can be executed in hybrid environments, combining real and simulated entities. This abstraction layer also allows a transparent simulation of these applications. Thus, a tested application can

then be deployed in a real environment without requiring any modification on its application code.

Part II

DEVELOPING PERVASIVE COMPUTING APPLICATIONS

3

DESIGNING AN APPLICATION

The first stage of our tool-based methodology is to design a pervasive computing application. To this end, we provide a design language dedicated to describing pervasive computing applications: the DiaSpec design language. DiaSpec first allows to describe the entities that compose a pervasive computing area. It also provides an ADL layer to architect the components of a pervasive computing application.

3.1 DESIGNING THE TAXONOMY

To cope with the growing number of application areas, DiaSpec [1] offers a taxonomy language dedicated to describing classes of entities that are relevant to the target application area. An entity consists of sensing capabilities, producing data, and actuating capabilities, providing actions. Accordingly, an entity description declares a data source for each one of its sensing capabilities. As well, an actuating capability corresponds to a set of method declarations. An entity declaration also includes attributes, characterizing properties of entity instances (*e.g.,* location, accuracy, and status). Entity declarations are organized hierarchically allowing entity classes to inherit attributes, sources, and actions.

An extract of the DiaSpec taxonomy for the heating control system is shown in Figure 4. Entity classes are introduced by the **device** keyword. Note that the same keyword is used to introduce both software and hardware entities.

To distinguish entity instances, attributes are introduced using the **attribute** keyword. Attributes are used as area-specific values to discover entities in a pervasive computing environment. They also allow the tester and the system administrator to discriminate entity instances during the simulation and deployment phases. For example, hardware entities of our taxonomy extend the abstract `LocatedDevice` entity that introduces the `location` attribute.

The sensing capabilities of an entity class are declared by the **source** keyword. For example, the `MotionDetector` entity defines the `detection` data source (line 6). Sometimes, retrieving a data

1. The DiaSpec grammar can be found at `http://diasuite.inria.fr/`

source requires a parameter. For example, the `schedule` data source of the `Calendar` entity maps an occupancy schedule to a room. In this case, the source needs to be parameterized by a location (line 18). Such parameters are introduced by the **indexed by** keyword.

Actuating capabilities are declared by the **action** keyword. As an example, the `Heater` declaration defines the `Heat` action interface. This action may be invoked by an application to start or stop the heating (line 14). The `Heat` interface is defined independently in lines 24 to 27.

```
1  device LocatedDevice {
2    attribute location as Location;
3  }
4
5  device MotionDetector extends LocatedDevice {
6    source detection as Boolean;
7  }
8
9  device TemperatureSensor extends LocatedDevice {
10   source temperature as Temperature;
11 }
12
13 device Heater extends LocatedDevice {
14   action Heat;
15 }
16
17 device Calendar {
18   source schedule indexed by location as Location;
19 }
20
21 structure Location { room as String; }
22 structure Temperature { value as Float; }
23
24 action Heat {
25   on();
26   off();
27 }
28
29 [ ... ]
```

Figure 4: Extract of the heating control system taxonomy. DiaSpec keywords are printed in **bold**.

The taxonomy layer of DiaSpec is domain specific in that it offers constructs corresponding to concepts that are essential to the pervasive computing domain. This is illustrated by the **source** and **action** constructs that directly correspond to the sensing and actuating concepts. As such, our taxonomy layer offers an abstraction layer between the entity implementation and the application logic. Indeed, on the one hand, the entity developer takes an entity declaration as a specification to which an entity implementation must conform. On the other hand, the applica-

tion architect can construct its specification on top of this set of entity declarations, abstracting over the heterogeneity of these entities.

We now present the architectural layer of the DiaSpec language, which is built on top of this taxonomy layer.

3.2 ARCHITECTING THE APPLICATION

The DiaSpec language provides an ADL layer to define application architectures. This layer is dedicated to the Sense/Compute/Control architectural pattern. This architectural pattern is commonly used in the pervasive computing domain [17, 21]. It consists of *context components* fueled by sensing entities. These components process gathered data to make them amenable to the application needs. Context data are then passed to *controller components* that trigger actions on entities.

Following this architectural pattern, the ADL layer of DiaSpec allows the context and controller components to be defined and the corresponding data-flow to be specified. Their definition depends on a given taxonomy, specified in the previous step of our methodology. Describing the application architecture allows to further specify a pervasive computing application, making explicit its functional decomposition.

We illustrate the ADL layer of DiaSpec with our heating control system. The overall architecture of this application is displayed in Figure 5 and all components are described in Table 1. At the bottom of this figure are the entity sources, as described in the taxonomy. The layer above consists of the context components fueled by entity sources. These components filter, interpret, and aggregate these data to make them amenable to the application needs. Above the context layer are the controller components that receive application-level data from context components and determine the actions to be triggered on entities. At the top of Figure 5 are the entity actuators receiving actions from controller components.

The DiaSpec description of the architecture of the heating control system is presented in Figure 6. In this application, temperature values are provided to the `AverageTemperature` component, declared using the **context** keyword. This component calculates the average temperature for each room of a building. It processes the average temperature using the `temperature` source provided by the temperature sensors. This is declared using the **source** keyword that takes a source name and a class of entities. To process the average temperature on a per-room basis, this context is declared as indexed by `Location`. In doing so,

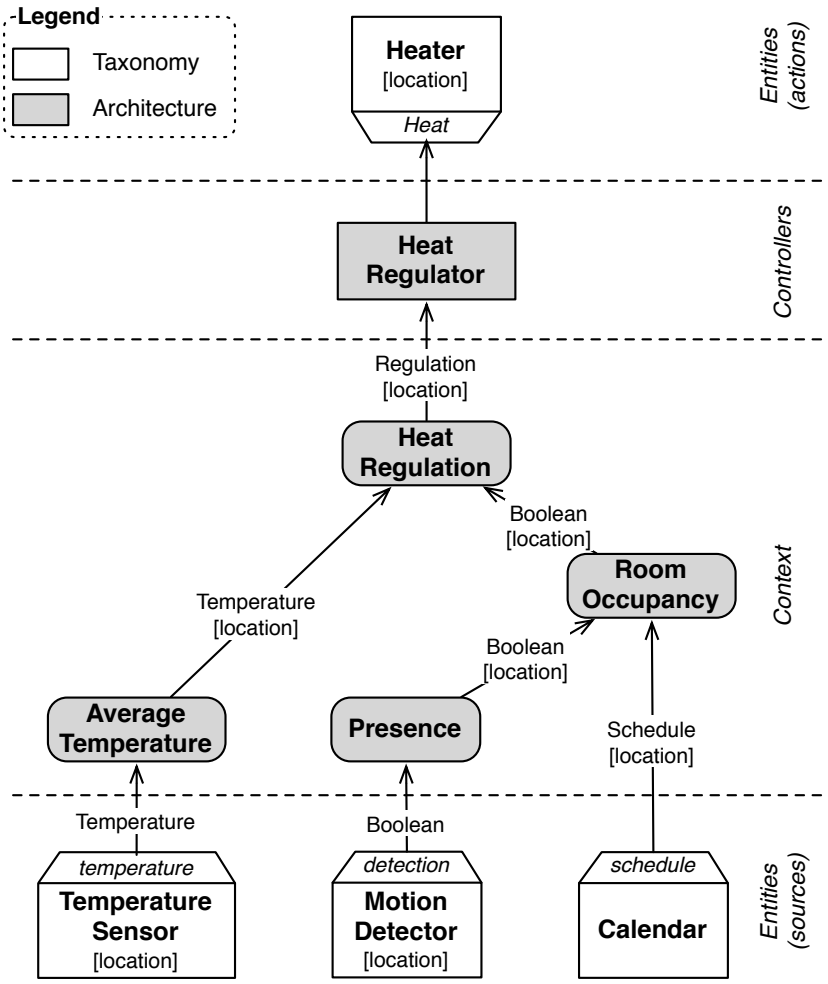

Figure 5: Specification of the heating control system.

each calculated average temperature is associated with a location. The `Presence` context determines whether a room is currently occupied from the information provided by motion detectors. The `RoomOccupancy` context determines a more advanced room occupancy than the `Presence` context. This occupancy takes into account the information provided by the `Presence` context, as well as the room schedule given by a calendar. Thus, the information provided by `RoomOccupancy` allows to heat a room prior to being occupied. When the occupancy of a room changes, the `HeatRegulation` context is invoked. Depending on the current temperature in this room, it may order a heat regulation to the `HeatRegulator` controller, declared using the **controller** keyword. The controller acts on `Heater` instances to regulate the temperature as required by the `HeatRegulation` context. This is declared using the **action** keyword.

The heating control system architecture illustrates the domain-specific nature of the DiaSpec ADL, providing the developer with pervasive computing concepts. These concepts are high

Type	Component	Responsibility
Entity *(sensing)*	`TemperatureSensor`	Provides the current temperature.
	`MotionDetector`	Detects motion in a room.
	`Calendar`	Provides the occupancy schedule of each room.
Context	`AverageTemperature`	Computes the average temperature of each room of the building.
	`Presence`	Aggregates the motion detection events and notifies if a room is occupied.
	`RoomOccupancy`	Notifies if a room is occupied depending on its occupancy schedule and its actual occupancy status.
	`HeatRegulation`	Notifies if a room needs to be heated.
Controller	`HeatRegulator`	Controls heaters to start or stop heating the rooms of the building.
Entity *(actuating)*	`Heater`	Heating system located in every room of the building.

Table 1: Components of the heating control system.

```
1  context AverageTemperature as Temperature indexed by location as
       Location {
2    source temperature from TemperatureSensor;
3  }
4
5  context Presence as Boolean indexed by location as Location {
6    source detection from MotionDetector;
7  }
8
9  context RoomOccupancy as Boolean indexed by location as Location {
10   source schedule from Calendar;
11   context Presence;
12 }
13
14 context HeatRegulation as Regulation indexed by location as
       Location {
15   context AverageTemperature;
16   context RoomOccupancy;
17 }
18
19 controller HeatRegulator {
20   context HeatRegulation;
21   action Heat on Heater;
22 }
```

Figure 6: Architecture of the heating control system. DiaSpec keywords are printed in **bold**.

level, making an architecture description concise and readable. It represents a useful artifact to share with application developers and other stakeholders. Moreover, the DiaGen code generator turns the role of this artifact from contemplative to productive, guiding the implementation of the declared components.

4

IMPLEMENTING AN APPLICATION

DiaGen automatically generates a Java programming framework from both a taxonomy definition and an architecture description. After outlining the implementation of DiaGen, we briefly present a generated programming framework. This presentation is then used to explain how a developer implements entities and the application logic on top of that framework.

4.1 PROGRAMMING FRAMEWORK GENERATOR

DiaGen generates a Java programming framework with respect to a taxonomy definition and an architecture description. DiaGen follows the design of typical code generators. As illustrated in Figure 7, there are three main phases: (1) the parser, (2) the type checker, and (3) the code generator.

The parser relies on the ANTLR [2] parser generator. Using a parser generator allows to easily refine/extend the DiaSpec language. The resulting Abstract Syntax Tree (AST) is then type-checked, ensuring for example that the inter-component communications conform to the architectural style (*e.g.,* a controller cannot communicate directly with the source of an entity). The type-checker is implemented in Java, using visitors. Finally, the code generator is in charge of producing the programming framework from the AST. The generator is written using the StringTemplate [66] engine. StringTemplate is a Java template engine for generating source code, web pages, or any other formatted text output.

4.2 GENERATED PROGRAMMING FRAMEWORK

A generated programming framework contains an *abstract class* for each DiaSpec component declaration (entity, context, and

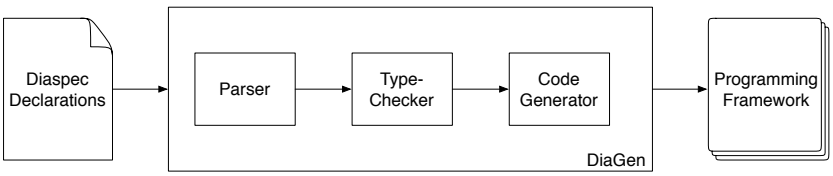

Figure 7: Structure of the DiaGen compiler.

controller) that includes generated methods to support the implementation (*e.g.*, entity discovery and interactions). The generated abstract classes also include abstract method declarations to allow the developer to program the application logic (*e.g.*, triggering entity actions).

Implementing a DiaSpec component is done by *sub-classing* the corresponding generated abstract class. In doing so, the developer is required to implement each abstract method. The developer writes the application code in subclasses, not in the generated abstract classes. As a result, in our approach, one can change the DiaSpec description and generate a new programming framework without overriding the developer code. The mismatches between the existing code and the new programming framework are revealed by the Java compiler.

A generated programming framework also contains *proxies* to allow entities to be distributed over the network. This is complemented by interfaces that allow the developer to interact with entities transparently, without dealing with the distributed systems details. Finally, a programming framework contains high-level support to manipulate sets of entities easily, following the Composite design pattern [31].

We now describe the process of implementing an entity, a context component, and a controller component by leveraging a generated programming framework. Along the way, we explain how the developer is guided by a dedicated programming framework.

4.3 IMPLEMENTATION OF ENTITIES

The compilation of an entity declaration in the taxonomy produces a dedicated skeleton in the form of an abstract class depicted in Figures 8 and 9. We now examine the generated support for each part of an entity declaration: attributes, sources, and actions.

ATTRIBUTES Entities are characterized by attributes. These attributes can be assigned values at runtime. Attributes are managed by generated getters and setters in the abstract class. For example, the `MotionDetector` entity has a `location` attribute (inherited from `LocatedDevice`, Figure 4, line 2) that triggers the generation of an implemented `setLocation` method (Figure 8, line 12). In each subclass, the developer will use the `setLocation` method to set the location of the motion detector. The initial value for each attribute must be passed to the generated constructor (Figure 8, line 4).

```
// from line  5
public abstract class AbstractMotionDetector {

  protected AbstractMotionDetector (Location location) {
     super(...);
     setLocation(location);
  }

  // from LocatedDevice, line 2
  private Location location;
  public Location getLocation() {return location;}
  protected void setLocation(Location location) {...}

  // from MotionDetector, line 6
  protected void setDetection(Boolean detection) {...}

  [ ... ]
}
```

Figure 8: The Java abstract class `AbstractMotionDetector` generated by DiaGen from the declaration of the MotionDetector entity (Figure 4, lines 5 to 7).

```
// from line  13
public abstract class AbstractHeater {

  protected AbstractHeater (Location location) {
     super(...);
     setLocation(location);
  }

  // from LocatedDevice, line 2
  private Location location;
  public Location getLocation() {return location;}
  protected void setLocation(Location location) {...}

  // from Heater, line 25
  public abstract void on();

  // from Heater, line 26
  public abstract void off();

  [ ... ]
}
```

Figure 9: The Java abstract class `AbstractHeater` generated by DiaGen from the declaration of the Heater entity (Figure 4, lines 13 to 15).

SOURCES An entity source produces values for context components. Support for this propagation is generated by DiaGen, allowing the entity developer to invoke these methods to fuel this process. For example, from the `MotionDetector` declaration and its data source (Figure 4, line 6), the generated abstract class (Figure 8) implements the `setDetection` (line 15) method. This method is to be called by a motion detector implementation.

ACTIONS An action corresponds to a set of operations supported by an entity. It takes the form of a set of abstract methods implemented by the abstract class generated for an entity declaration. Each operation is to be implemented by the entity developer in the subclass. This implementation bridges the gap between the declared interface and an actual entity implementation. For example, the generated `Heater` abstract class (Figure 9) declares the abstract methods `on` and `off` (lines 15 and 18) that need to be implemented in all subclasses.

The code fragment in Figure 10 is an implementation of a `MotionDetector` entity that uses a camera to detect motion. The `CameraMotionDetector` implementation of the motion detection relies on a third-party library that interfaces networked cameras. When a motion is detected by the third-party library (Figure 10, line 13), the developer uses the `setDetection` method provided in the generated abstract class `AbstractMotionDetector` (Figure 8, line 15).

4.4 DEVELOPING THE APPLICATION LOGIC

The implementation of a context or controller component also relies on generated abstract classes. The development of the application logic thus consists of sub-classing the generated abstract classes.

4.4.1 *Implementation of Context Components*

From a context declaration, DiaGen generates programming support to develop the context processing logic. The implementation of a context component processes input data to produce refined data to its consumers. The input data are either pushed by an entity source or pulled by the context component. Both modes are provided to the developer for each source declaration of the architecture.

The code fragment in Figure 11 presents the implementation of the `AverageTemperature` context (from Figure 6, lines 1 to 3). This

```
public class CameraMotionDetector extends AbstractMotionDetector
    implements MotionDetectionListener  {

  private Camera camera;

  public CameraMotionDetector(Location location) {
     super(location);
     camera = new AxisCamera(''cam3.bordeaux.inria.fr'');
     camera.addMotionDetectionListener(this);
  }

  // from the MotionDetectionListener interface.
  // Called by Camera when a motion is detected
  @Override
  public void motionDetected(Camera camera) {
    setDetection(true);
  }

}
```

Figure 10: A developer-supplied Java implementation of a `MotionDetector` entity. This class extends the generated abstract class shown in Figure 8. The implementation relies on a third party library: `motionDetected` is a callback method from the `MotionDetectionListener` interface.

is done by extending the corresponding generated abstract class named `AbstractAverageTemperature`. The value provided by this context is only pulled by the `HeatRegulation` context. Thus, the developer only needs to override the method that is called when the value is pulled, namely the `getAverageTemperature` method. The developer is provided with a `location` index. This allows this context to provide the average temperature for a specific location. The developer is also provided with a `Discover` object. This object enables to discover the temperature sensors located in a specific location. It also enables to pull the temperature data for these temperature sensors, using the `getTemperature` method. Finally, when the average temperature is calculated, the developer returns this value. The generated framework takes care of returning this value to the component that required it.

4.4.2 *Implementation of Controller Components*

A controller component differs from a context component in that it takes decisions that are carried out by invoking entity actions. A controller declaration explicitly states which entity actions it controls. This information is used to generate an abstract class for each controller component. This abstract class provides support for discovering target entities and for invoking their actions. From the declaration of the heat regulator

```
1  public class AverageTemperature extends AbstractAverageTemperature
       {
2
3    private static float DEFAULT_TEMPERATURE = 20;
4
5    @Override
6    public Temperature getAverageTemperature(Location location,
         Discover discover) {
7      TemperatureSensorComposite sensors =
           discover.temperatureSensorsWhere().location(location);
8      if (sensors.size > 0) {
9        int sumTemperature = 0;
10       for (TemperatureSensor sensor : sensors) {
11         sumTemperature += sensor.getTemperature().getValue();
12       }
13       float averageTemperature = sumTemperature / sensors.size();
14       return new Temperature(averageTemperature);
15     } else
16       return new Temperature(DEFAULT_TEMPERATURE);
17   }
18
19 }
```

Figure 11: A developer-supplied implementation of the `AverageTemperature` context.

(Figure 6, line 19), DiaGen generates an abstract class named `AbstractHeatRegulator`. Figure 12 shows an implementation for this controller. When the `HeatRegulator` context produces a new value, the `onHeatRegulation` method is invoked (line 4) in the `HeatRegulator` implementation. The method starts by discovering heaters present in a given location (line 5). It then turns on or off these heaters depending on the received heat regulation by invoking the remote methods `on` or `off`. This ability to discover and command heaters from the heat regulator comes from the architecture declaration (Figure 6, line 21).

```
public class HeatRegulator extends AbstractHeatRegulator {                1
                                                                           2
  @Override                                                                3
  public void onHeatRegulation(HeatRegulation regulation, Discover         4
      discover) {
    HeaterComposite heaters =                                              5
        discover.heatersWhere().location(regulation.getLocation());
    if (regulation.getType() == Regulation.START_HEATING)                  6
      heaters.on();                                                        7
    else if (regulation.getType() == Regulation.STOP_HEATING)              8
      heaters.off();                                                       9
  }                                                                        10
                                                                           11
}                                                                          12
```

Figure 12: Implementation of the `HeatRegulator` controller.

After having presented the programming support given by a generated framework, we focus on a key mechanism to cope with dynamicity, namely, entity discovery.

4.5 ENTITY DISCOVERY

Our dedicated programming framework provides support to discover entities based on the taxonomy definition. Entity discovery returns a collection of proxies for the selected entities. This collection is encapsulated in a *composite object* that gathers a collection of entities [31]. An example of such collection, `HeaterComposite`, is returned in line 5 of Figure 12. Thanks to this design pattern, the developer can process all elements of the collection either explicitly by using a loop, or implicitly by invoking a method of the composite, which is part of the generated programming framework. Lines 7 and 9 in Figure 12 is an example of an implicit iteration.

To help developers express queries to discover entities, DiaGen generates a Java-embedded, type-safe Domain-Specific Language (DSL), inspired by the work of Kabanov *et al.* [39] and by fluent interfaces introduced by Fowler [29]. Existing works often use strings to express queries, which defer to runtime the detection of errors in queries. In our approach, the Java type checker ensures that the query is well formed at compile time. This strategy contrasts with other works where the Java language is augmented, requiring changes in the Java compiler and integrated development environments, as illustrated by Silver [74] and ArchJava [1].

A method suffixed by `Where` is available for each device that can be discovered. These methods return a dedicated filter object on which it is possible to add filters over attributes associated with the entity class. For example, the `HeatRegulator` abstract class defines a `heatersWhere` method that returns a `HeaterFilter`. This filter can be refined by adding a filter over the `location` attribute inherited by the `Heater` in the taxonomy. This is done by calling the `location()` method defined in the generated `HeaterFilter` class. The parameter to this method is either a `Location` value or a logical expression. If a `Location` value is passed, the discovered entities are those with an `location` attribute equals to the passed value. An example of the use of a value is given in the `HeatRegulator` class shown in Figure 12. The `onHeatRegulation` method selects heaters to operate. The call to `heatersWhere` (line 5) restricts the selection to screens located in the area where the news should be published.

If a logical expression is chosen, the attributes of the selected entities hold with respect to the logical expression. A logical ex-

pression is made of relational and logical operators. For example, the following query selects screens that are either located in room 1 or 2:

```
Location room1, room2;
...
discover(
 screensWhere().area(or(eq(room1),eq(room2)))
);
```

This embedded DSL is both expressive and concise. It plays a key role in enabling the developer to handle the dynamicity of a pervasive computing environment without making the code cumbersome.

Part III

TESTING PERVASIVE COMPUTING APPLICATIONS

5

TESTING AN APPLICATION WITH DIASIM

To address the testing stage, we propose an approach and a tool integrated in our tool-based methodology, namely DiaSim. Our approach uses the testing support generated by DiaSpec to transparently test applications in a simulated physical environment. In this chapter, we first describe the simulation model underlying our simulation approach. Then, we present how to develop a simulation using the testing support generated by DiaSpec. Finally, we detail how applications are tested with DiaSim.

5.1 SIMULATION MODEL

Let us now describe the key concepts of our approach to simulating a pervasive computing system.

5.1.1 *Stimulus Producers*

Stimuli are changes of the environment that are observed by the sensors of the pervasive computing environment. From a simulation perspective, emitting environment stimuli may trigger an entity data source (*e.g.,* the detection source of a motion detector) that publishes events, that may eventually trigger actions on actuators (*e.g.,* turning on a light). Emitters of stimuli are called *stimulus producers*; they are dedicated to a type of stimulus.

Every stimulus has a type that matches the type of one or more data sources provided by entities. Additionally, every type of stimulus is associated with a set of rules defining its evolution in terms of space, time and value. Physical environment stimuli are often modeled by mathematical definitions (*e.g.,* with ordinary/partial differential equations). Such a definition is typically provided by experts of the application area or the literature in related fields. For example, temperature stimuli required for testing a heating control system can be modeled with heat transfer formulas described in any thermodynamics books (*e.g.,* [41]).

Other types of stimulus can be introduced by replaying logs of measurements collected in a real environment. For example, to design zero-energy building, extensive measurements are carried out to log the variations in temperature, light and wind over a one-year period [30]. This line of work contributes to building

a rich library of measurements, facilitating simulation without compromising accuracy.

However, measurement logs are not available in general for simulation (*e.g.,* fire simulation), requiring the definition of some model to approximate a real environment, as accurately as necessary. To achieve this goal, our approach is to define an approximation model with respect to each type of stimulus managed by the sensors of an environment. For example, the simulation of location-related sensors can be defined as processing Cartesian coordinate stimuli. If location-related sensors report location information at the granularity of a room, coarse-grain information can be generated by the stimulus producers (*e.g.,* a unique Cartesian coordinate stimulus per room).

Because a type of stimulus can be consumed by different entity data sources, stimulus producers are decoupled from the simulated entities.

So far, we described stimuli as being directly processed by entities. However, a type of stimulus can also influence the evolution of other types of stimulus; such a type of stimulus is called a *causal stimulus*. For example, fire could be declared as a causal stimulus if we needed to model its resulting action on the temperature stimulus. When a stimulus does not impact others, it is called *simple stimulus*.

5.1.2 *Simulated Entities*

A simulated environment consists of stimulus producers and simulated entities. Like a real entity, a simulated entity interacts with a simulated environment by processing stimuli, performing actions, and exchanging data with pervasive computing applications. An entity has two kinds of facets, each one playing a key role in simulation: data source and action. The simulated version of a data source mimics the behavior of a real data source, reacting to stimuli generated by the stimulus producers. For example, the simulated version of a motion detector, when turned off, ignores coordinate stimuli. Otherwise, when the motion detector is on and receives coordinate stimuli matching its room, a motion event is published with its room identifier.

An action provided by an entity typically modifies the state of this entity as well as the observable environment context. For example, invoking an action on a light to turn it on, changes the light state and locally increases the luminosity. The simulated version of a light thus needs to maintain its state (on/off) and to create a stimulus producer to increase luminosity with respect to an intensity specific to the light.

In addition to defining their simulated versions, entities need to be deployed. For example, the simulated `Light` entity needs to be instantiated as many times as required to mimic the real environment. In doing so, entity instances may be assigned specific attribute values such as their location and luminosity intensity in the light example.

As for context and controller components, they are insensitive to whether or not entities are simulated. For example in our heating control system, the same implementation of a `HeatRegulator` controller operates `Heater` instances, regardless of whether or not they are simulated. As well, the `Presence` context will not interact any differently with a simulated or a real `MotionDetector` instance.

5.1.3 *Physical Space*

To complete the simulation of an environment, we need to model the physical space (*e.g.*, an office space, an apartment, a building or a campus) and to make it evolve as the simulation scenario unfolds. A simulated space allows us to model stimulus propagation, according to pre-defined rules. As well, it is annotated with the location for each entity instance whose real version may impact the physical environment, whether they are fixed, mobile and dynamically appearing.

The model of a physical space is decomposed into polygon-shaped regions. This decomposition is hierarchical, breaking down a physical space into increasingly narrow regions. For example, a building consists of floors, each of which has corridors and rooms, *etc*. Entity instances are positioned in the simulated space, in accordance to the desired (or existing) physical setting to be simulated. As an approximation, the intensity of a stimulus is assumed to be uniform within a region.

Our overall simulation model is depicted in Figure 13. Stimulus producers emit stimuli of various types according to a scenario. The values of the stimuli can either be read from logs of measurements or can be computed from an approximated physical model. In place of data sources of real entities, data sources of simulated entities process these stimuli and produce events. The unchanged application reacts to these events by invoking entity actions. In turn, actions change the simulated environment, triggering stimulus producers.

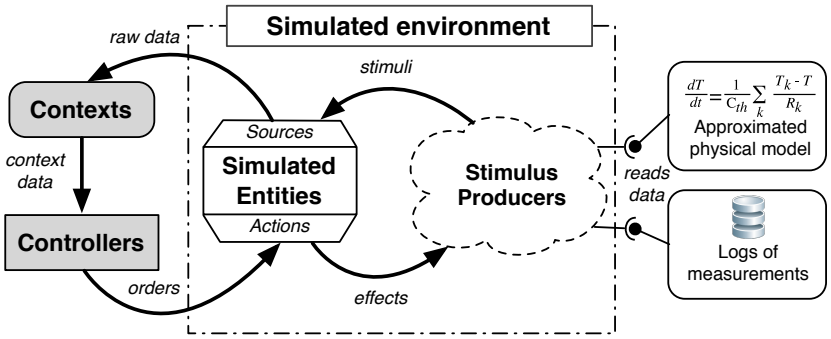

Figure 13: Simulation model.

5.2 DEVELOPING A SIMULATION

Given the simulation model presented earlier, we are now ready to develop the simulated version of entities and stimulus producers, forming a simulation scenario.

5.2.1 *Developing Simulated Entities*

As presented in Chapter 4, the DiaSpec compiler generates a dedicated programming framework from a DiaSpec description to develop real entities. Besides this generated programming framework, the DiaSpec compiler generates a simulation programming framework to develop simulated entities. For each entity class, a set of Java classes is generated for programming real and simulated entities, as depicted in Figure 14: real entities (*e.g.*, $\mathcal{R}_1$) extend the $\mathcal{C}$ abstract class of the real programming framework, whereas simulated entities (*e.g.*, $\mathcal{S}_2$) extend the $\mathcal{C}'$ abstract class of the simulation programming framework. A simulation programming framework inherits support provided by the related real programming framework and adds simulation-specific functionalities. For instance, it enables entities to receive simulated stimuli and to trigger stimulus producers. Figure 15 shows a generated abstract class that is used for implementing simulated motion detectors. To implement the simulated version of an entity, the tester subclasses the corresponding abstract class. For instance, Figure 16 shows the implementation of a simulated motion detector named `MySimulatedMotionDetector`. The related `SimulatedMotionDetector` abstract class contains an abstract method to receive simulated detection events (`receive`) and a concrete method to publish `MotionDetection` events (`publish`).

As illustrated by Figure 16, the implementation of a simulated entity is often trivial. It only forwards the received stimuli. Thus,

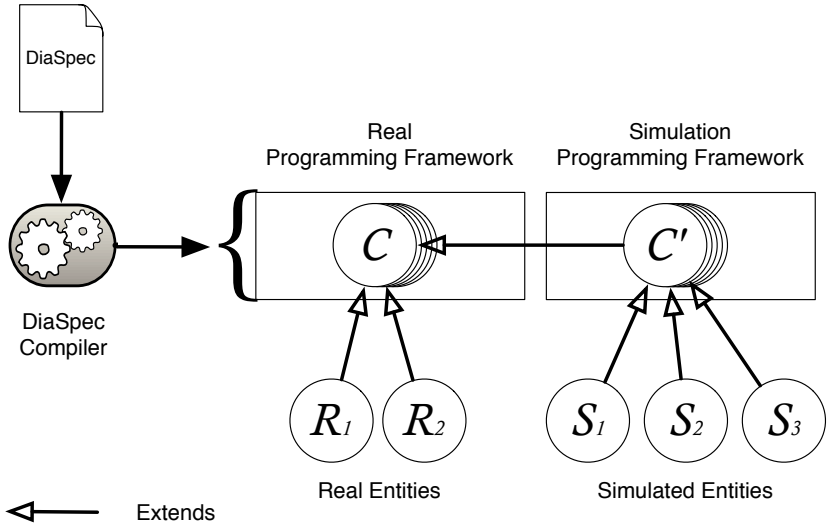

Figure 14: Correspondence between real and simulation programming frameworks.

to simplify the tester task, the simulation layer of the generated programming framework provides such implementations for all the simulated entities. However, nothing prevents the tester from implementing more sophisticated behaviors by extending the corresponding abstract class.

```
1  public abstract class SimulatedMotionDetector extends
2                      AbstractMotionDetector implements
                           SimulatedEntity {
3
4    public SimulatedMotionDetector(Location location) {
5       super(location);
6    }
7
8    @Override
9    public void receive(Stimulus stimulus) {
10      if (stimulus.getName().equals("detection")
11         receive((Boolean)stimulus.getEvent());
12   }
13
14   public abstract void receive(Boolean detection);
15 }
```

Figure 15: Implementation of the generated `SimulatedMotionDetector` class.

5.2.2 *Developing Hybrid Environments*

Our approach permits real entities to be used in a simulated environment, whenever desirable. This key feature enables real entities to be incrementally added to the simulated environment,

```
1  public class MySimulatedMotionDetector extends
2                      SimulatedMotionDetector {
3
4    public MySimulatedMotionDetector(Location location) {
5       super(location);
6    }
7
8    @Override
9    public void receive(Boolean detection) {
10      publish(detection);
11   }
12 }
```

Figure 16: Implementation of a simulated `MotionDetector` entity

facilitating the transition to a real environment. Also, this strategy enables to improve the rendering of a simulation by mixing real entities. For example, a real LCD screen can be introduced in a simulation to display messages that future users will read.

To examine how real entities are integrated in a simulated environment, recall our inheritance strategy, as illustrated in Figure 14. Because of this strategy, when a controller looks up a given entity type, it receives the real entities, as well as the simulated ones. Similarly, when a context subscribes to a data source, it can interact with both real and simulated data sources. This approach allows applications to be executed in a hybrid environment. Furthermore, real and simulated entities can be added dynamically, as the simulation of a pervasive computing system runs.

5.2.3 *Developing Stimulus Producers*

The development of stimulus producers is facilitated by a simulation programming framework. This programming support provides a generic `StimulusProducer` class that the tester can use to create his own stimulus producers. Classes of stimulus are defined from types of data sources defined in DiaSpec. For example, the building management area includes stimuli of temperature and motion detection. Several stimulus producers can be attached to the same class of stimulus. For example, if a room contains two heaters, each one has its own producer of temperature stimuli. A stimulus producer defines the evolution of a source of stimuli. For example, to simulate fire gaining intensity, a stimulus producer gradually increases the intensity of the emitted fire stimulus.

In our heating control system, we use this simulation programming framework to produce motion events when simulated peo-

```
public class MyAgentModel extends DiaSimAgentModel implements
    AgentListener {

  private static int RANGE = 5;
  private StimulusProducer stimulusProducer;

  public MyAgentModel(World world) {
    super(world);
    Source motionDetectionSource = new Source("MotionDetector",
        "detection", "Boolean");
    stimulusProducer = new
        StimulusProducer(motionDetectionSource);
  }

  @Override
  public List<DiaSimAgent> createAgents() {
    List<DiaSimAgent> agents = super.createAgents();
    AgendaStimulusProducer studentAgenda = new
        AgendaStimulusProducer(``resources/studentagenda.xml'');
    AgendaStimulusProducer teacherAgenda = new
        AgendaStimulusProducer(``resources/teacheragenda.xml'');
    for (DiaSimAgent a : agents) {
      agent.addAgentListener(this);
      if (agent.getType().equals(``Student'')
        agent.setAgendaStimulusProducer(studentAgenda);
      else if (agent.getType().equals(``Teacher'')
        agent.setAgendaStimulusProducer(teacherAgenda);
    }
    return agents;
  }

  @Override
  public void agentMoved(Agent agent, String location) {
    for (DiaSimDevice d : getDevices()) {
      int distance = agent.distanceFrom(d.getPosition());
      if (d.getType().equals("MotionDetector")
              && distance < RANGE)
        stimulusProducer.publish(true,location);
    }
  }
}
```

Figure 17: Implementation of the `MyAgentModel` class used in the heating control system simulation. This class is responsible for publishing motion detection events when simulated people come within a range of a motion detector.

ple move in the range of a motion detector. To illustrate the use of the simulation programming framework, Figure 17 presents the implementation of the class that publishes motion events. This class extends `DiaSimAgentModel`. The `DiaSimAgentModel` class is provided by the simulation programming framework and provides programming support for handling the simulated people of the simulation. In this example, it is used to be notified when a simulated agent moves in the detection area of a motion detector. A stimulus producer is created in this class: `stimulusProducer` (Figure 17, line 9). The simulation programming framework allows to be notified when an agent moves by implementing the `AgentListener` interface. When an agent moves, the `agentMoved` method is called (Figure 17, line 28). Finally, when an agent moves in the detection area of a motion detector, a motion detection stimulus is published (Figure 17, line 33).

Pervasive computing systems often interact with people. For instance, our heating control system relies on the detection of motion. To help introducing the behavior of simulated people, we provide a class for defining the movements of the simulated agents during the simulation: `AgendaStimulusProducer`. This class is parameterized by an agenda describing where a simulated agent will be located during the simulation (Figure 17, lines 15 and 16). This agenda allows to define time slots during which the agent is in a specific location. This agenda is defined in XML. Figure 18 presents an extract of the `studentagenda.xml` file used in the `MyAgentModel` class (Figure 17, line 15). A simulated agent can be associated with an `AgendaStimulusProducer` object (see for example Figure 17, lines 20 and 22). Thus, this simulated agent will automatically move during the simulation with respect to this agenda. Though using an agenda to model the human behavior is very limited, we can test a wide range of pervasive computing applications with this basic support. The large-scale simulation of engineering school presented in Chapter 6 simulates 200 people with this simple support.

To summarize the relationships between the classes introduced in this section, Figure 19 presents a class diagram of the implementation of the heat regulator simulation. In this example, every class except `TemperatureStimulusProducer` is provided to the tester either by the generated DiaSpec framework, the simulation programming framework, the generated emulation layer, or Siafu. The tester may modify the simulated entity implementations (*e.g.*, `MySimulatedHeater`) if he needs a more sophisticated behavior as the one provided by default. He may also modify the `MyAgentModel` class if he needs to send simulated stimuli triggered by simulated agents. For instance, sending a simulated detection stimulus when an agent is in the scope of a motion detector would be implemented in the `MyAgentModel` class. It is

```
<?xml version="1.0" encoding="UTF-8"?>
<agenda>
  <item>
    <location>I 112</location>
    <startTime>11/04/2011 10:30:00 GMT</startTime>
    <endTime>11/04/2011 11:50:00 GMT</endTime>
  </item>
  <item>
    <location>Hall</location>
    <startTime>11/04/2011 11:50:00 GMT</startTime>
    <endTime>11/04/2011 12:00:00 GMT</endTime>
  </item>
  [ ... ]
</agenda>
```

Figure 18: Extract of the `studentagenda.xml` XML file.

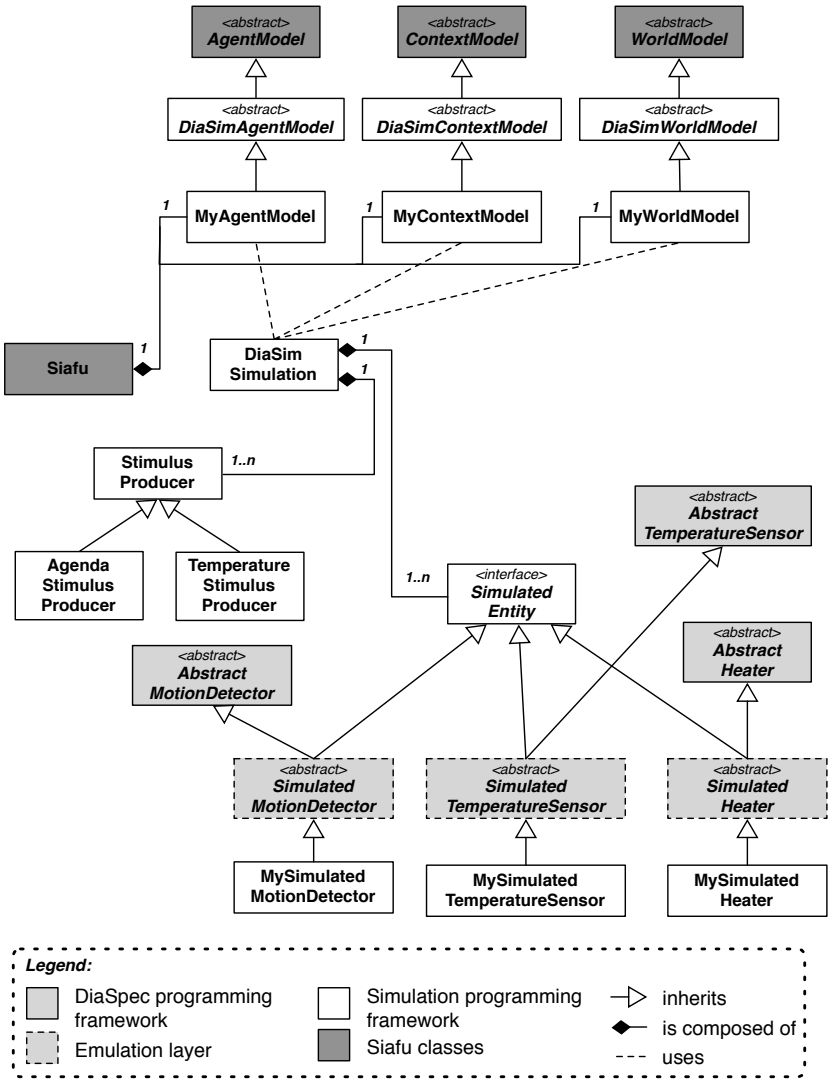

Figure 19: Class diagram of the implementation of the heat regulator simulation.

important to notice that the simulated entity implementations are independent from the stimulus producers. The communication between the stimulus producers and the simulated entities is handled by the `DiaSimSimulation` class. For instance, it is possible to modify the implementation of `TemperatureStimulusProducer` without modifying the `MySimulatedTemperatureSensor` implementation. Thus, this independence between stimulus producers and simulated entities would allow to compute temperature values from a thermal physical model instead of reading from logs of measurements without any impact on the simulated entity implementations.

5.3 TESTING SUPPORT

We now detail how applications are tested in the DiaSim simulator. DiaSim executes simulation scenarios, monitors simulations, and supports application debugging.

5.3.1 *Transparent simulation*

A programming framework generated by the DiaSpec compiler provides applications with an abstraction layer to discover entities. This entity discovery support is based on the taxonomy definition. In particular, it includes methods to select any node in the entity taxonomy. The result of this selection is the set of all entities corresponding to the selected node and its sub-nodes. The developer can further narrow down the entity discovery process by specifying the desired values of the attributes. This situation is illustrated in Figure 12. When a heat regulation is required in a particular location, the `HeatRegulator` controller implementation discovers the heaters located in this location and turns them on. The `discover` parameter is used to achieve this entity discovery. Using the value of `regulation.getRegulation()`, only the heaters in this particular location are discovered.

Because of this abstraction layer, simulation is achieved transparently: the same application code discovers and interacts with entities, whether or not simulated. This transparent simulation applies to all aspects of a pervasive computing application. For another example, simulated data sources can be added to a pervasive computing system, without requiring any change in the application code.

5.3.2 *Simulator architecture*

The overall architecture of DiaSim is displayed in Figure 20. It consists of an emulator to support the execution of pervasive computing applications and a simulator of context to manage stimuli. The simulator of context communicates the simulation data to the monitor for rendering purposes.

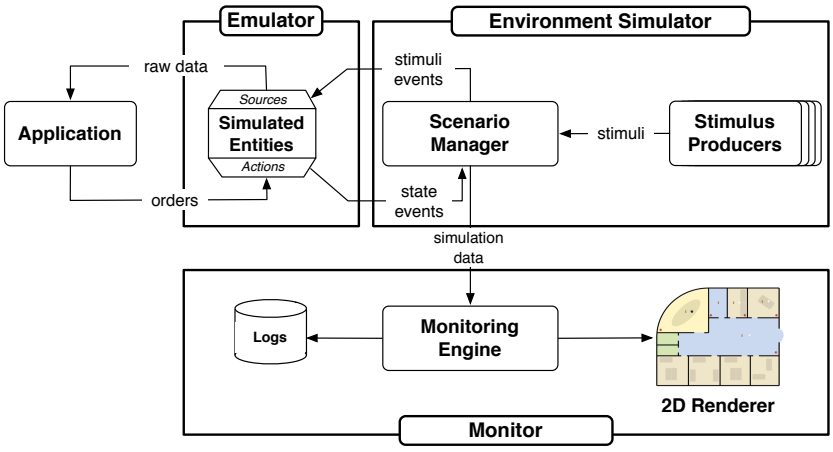

Figure 20: DiaSim architecture.

5.3.2.1 *Executing simulation scenarios*

An environment simulator generates stimuli as a given simulation scenario unfolds. It consists of stimulus producers and a scenario manager that dispatches stimuli to the relevant entities. The *scenario manager* is a mediator, periodically querying the stimulus producers to feed the data sources of simulated entities. For example, the scenario manager collects stimuli of outdoor luminosity and passes them to outside light sensors.

Actions can create changes to the simulated environment. To do so, entities register new stimulus producers to the scenario manager. For example, when fire is detected, a fire sprinkler discharges water on a given region. Because water is declared as a causal stimulus with respect to fire, it reduces the fire intensity. When the application deactivates the fire sprinkler, the water stimulus producer is stopped by the scenario manager.

5.3.2.2 *Monitoring simulation*

The scenario manager receives simulation data from stimulus producers and simulated entities to keep track of the simulated environment state. The scenario manager passes simulation data to the monitoring engine that graphically renders simulation scenarios. The monitoring engine also accepts live user interactions,

to pause the simulation or modify the scenario on-the-fly (*e.g.,* by adding new stimulus producers). Beyond the visual rendering of a simulation, we propose additional functionalities to DiaSim to further assist the user, as presented next.

5.3.3 *Application testing support*

Monitoring a simulation requires measuring, collecting and rendering a stream of simulation data. Because of its volume, simulation data often require to be approximated in order to be rendered. To do so, the simulated environment is approximated in space and time. Space approximation provides an idealized map of the physical space, rendering the evolution of simulated entities (*e.g.,* alarm ringing, motion detection) and stimuli (*e.g.,* fire spreading, people moving). Environments are also approximated in time, decoupling the rendering time from the real time. As a result, the user often cannot follow the simulation in real time. To focus on the sequence of events leading to an error, the monitoring engine of DiaSim provides time shifting functionalities, to replay part of a simulation. Raw data from the simulation log can be directly browsed by DiaSim, like network traces by network analyzers [27]. A simulation log contains information about interactions between entities (*i.e.,* time, source, destination, interaction parameters) and between stimuli and entities (*i.e.,* time, source, destination, class of stimuli, stimuli parameters). Replays help to isolate bugs but do not ensure applications have been fixed correctly. Reproducing exact testing conditions is required to validate a new version of an application. To do so, a simulation scenario completely defines the simulated environment and its behavior, making testing conditions deterministic and reproducible.

6

DIASIM IMPLEMENTATION

Our simulation approach has been implemented in the DiaSim tool. DiaSim is implemented in Java and consists of 15,000 lines of code. In this chapter, we first present the two pieces of software that compose DiaSim, namely the DiaSim scenario editor and the DiaSim simulation renderer. Then, we present a large-scale simulation that has been implemented to validate DiaSim.

6.1 IMPLEMENTATION

Simulating a pervasive computing application with DiaSim is realized in two steps: (1) editing a simulated physical environment and simulation scenarios, and (2) executing a simulation to test an application. To support these two steps, we developed two pieces of software: a scenario editor, and a simulation renderer. We describe these two tools in the remaining of this section.

6.1.1 *DiaSim Scenario Editor*

The first step to simulate a pervasive computing application is to model the physical environment. This model can be used to test multiple pervasive computing applications. The model of the physical environment is realized in a graphical editor. This editor allows to define the layout of a physical environment, including structural characteristics (*e.g.,* walls). Figure 21 shows the simulated school building that we modeled for testing the heating control system. In this example, an area has been defined for each room, corridor, and hall of the simulated school building. Then, using a DiaSim taxonomy, the tester defines and positions the simulated entity instances in the simulated physical environment. Figure 21 illustrates the configuration of simulated entity instances in the school building. In this example, simulated loudspeakers, screens, and badge readers are positioned in the main school hall. Simulated loudspeakers are also located in each corridor. Finally, simulated people are added to the simulation using the editor.

The second step of the simulation scenario definition is the configuration of stimulus producers. The DiaSim scenario editor supports the definition of stimulus producers and their behavior,

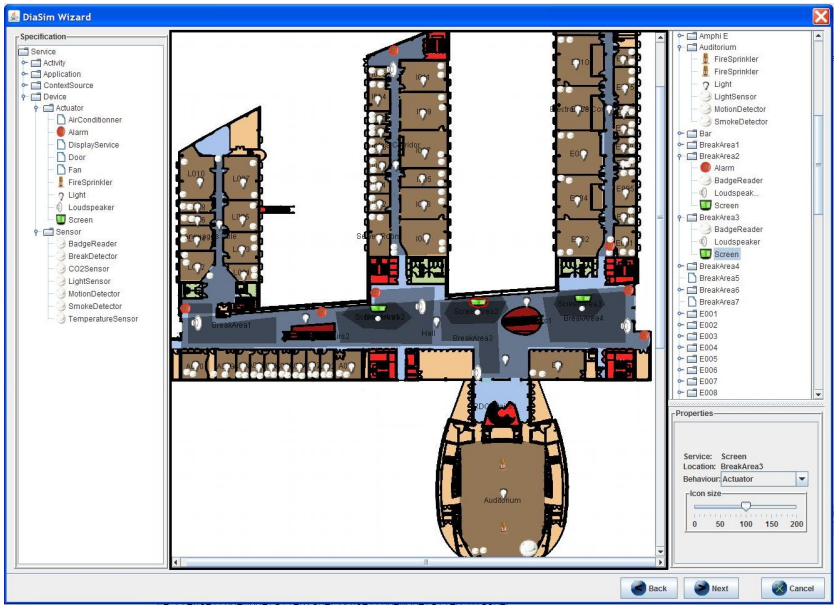

Figure 21: DiaSim scenario editor. The DiaSim editor is parameterized by an entity taxonomy. The entities defined in the taxonomy are displayed on the left panel of the graphical user interface. The entities can be dragged and dropped on the central panel to add simulated entity instances into the simulated environment.

by allowing the user to define stimulus intensities in areas of the simulated space at specific moments in time. For example, a producer of motion stimuli simulates a user moving in a school hallway at a given time. Alternatively, stimulus producers are defined by a modeling function (*e.g.*, a function defining the outside luminosity for 24-hour period) or previously logged measurements (*e.g.*, class schedules or statistics on class attendance).

Finally, the simulation scenario is saved as an XML file. This file can later be modified by the scenario editor.

6.1.2 *DiaSim Simulation Renderer*

The XML file of a scenario configures the DiaSim simulation renderer with the defined scenario. We studied numerous existing simulators for pervasive computing environments. We decided to use Siafu [45], a 2D-graphical context simulator. This choice was motivated by two key features: (1) Siafu provides a context simulation engine to model pervasive computing environments, and (2) Siafu is written in Java and could thus be easily interfaced with our tools. Thus, our simulator interfaces with Siafu to use its rendering and time-control functionalities. On top of a picture of the simulated space, the simulation renderer displays entities and stimuli, as shown in Figure 22.

The simulation renderer shows the state of the simulated entities, by displaying a bubble of raw text above entities (*e.g.*, when a data source publishes events) and/or modifying the visual representation of the entity (*e.g.*, a yellow light is displayed when turned on). To complement these macroscopic views, we enriched Siafu's rendering functionalities with Java and Web interfaces, and audio streams. In the ENSEIRB simulation, clicking on school LCD screens opens a Web interface showing what is currently displayed on the simulated screen. We also used this enriched programming support for simulating loudspeakers, allowing them to play audio streams.

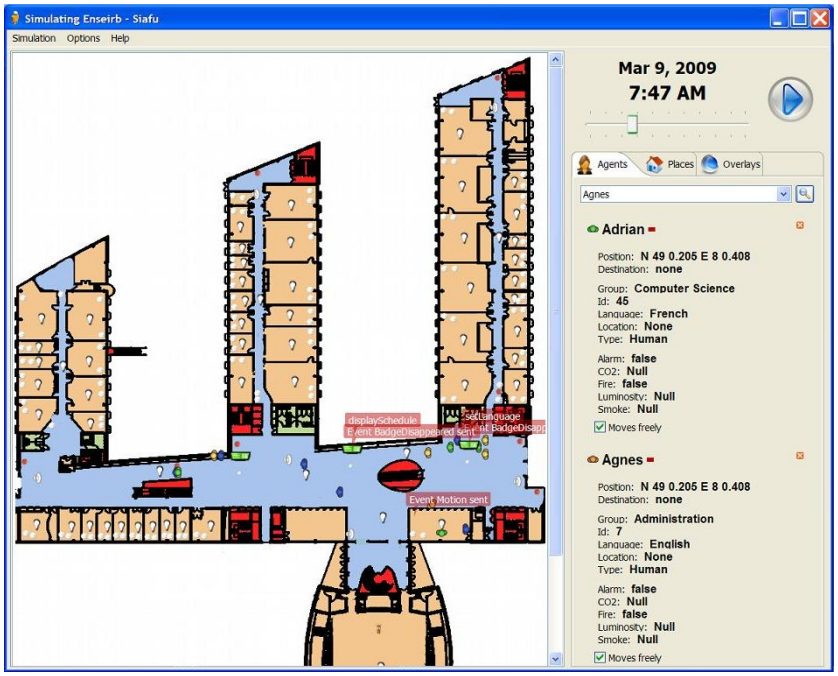

Figure 22: DiaSim simulation renderer. The simulated environment is displayed in the left part of the graphical user interface. The red popups transparently displayed above the simulated entities indicate that the entity has realized an interaction. More information about the simulated people and simulated entities can be found on the right of the graphical user interface.

DiaSuite supports several distributed systems technologies, including Java RMI [14], SIP [5] and Web Services. A back-end defines the communication protocol used by the DiaSpec components to communicate with each other. The simulation back-end used by DiaSim is derived from the Java RMI back-end. This strategy allows us to integrate remote real entities and to distribute the workload over several different hosts when numerous simulated entities are in play.

6.2 APPLICATIONS

We applied our tool-based methodology to a real-size case study: the management of a 13,500-square-meters building hosting an engineering school [38]. Six pervasive computing applications from different pervasive computing areas were developed for this case study. The heating control system is one of these applications. All these applications were simulated using DiaSim. In this simulation, over 110 entity instances and 200 occupants are simulated (*e.g.*, staff and faculty members, students, and visitors) with various behavorial pattern. We now briefly introduce these applications.

NEWSCAST Newscast aims to provide general information to users and to announce upcoming events with respect to their preferences; an example is given by Ranganathan *et al.* [57] for advertisement. This area requires devices to broadcast messages (*e.g.*, loudspeakers and screens). As well, users need to be identified to determine their preferences. This identification can be achieved by various means such as short-range badge readers. A variety of general and special-purpose information sources can be integrated in a Newscast application. For example, a source can consist of upcoming events. Another example of information source can be the status of the place where the Newscast application is run, enabling different Newscast policies (*e.g.*, holidays and workdays). In our case study, our Newscast application has two functionalities. It first announces the upcoming classes to the students using loudspeakers. Its second functionality is to display customized information to the students using screens positioned at various locations in the school building. The displayed pieces of information are the latest news about the school, as well as the class schedules. They are displayed with respect to the interests of the students standing near each individual screen. For example, the information displayed on a screen depends on the spoken languages, specialty, courses, and extracurricular activities of the students around it. The Newscast application detects the people surrounding a screen using the Bluetooth technology.

ANTI-INTRUSION The anti-intrusion application relates to the security area. The first functionality of this application is to turn on the alarms of the building when an intrusion is detected. An intrusion is detected using motion detectors located in every room and corridor of the building. When a motion is detected, the application verifies in the building occupancy schedule if the building is open or closed. Thus, if a motion is detected when the building is closed, the application turns on the alarms. The

second functionality of this application is to warn the building keeper by sending him an SMS when an intrusion is detected. This SMS enables the keeper to know the exact location of the intrusion.

ACCESS CONTROL The access control application also pertains to the security area. It controls the access to the building rooms that are secured by badge readers. The role of this application is to determine whether it unlocks the door when someone uses his badge for entering a room. Depending on the status of the badge owner (*e.g.,* student, teacher or building keeper) and the room (*e.g.,* classroom or server room), the application allows or refuses to unlock the door when someone uses his badge. The application retrieves the access control policies from a remote database managed by the security manager of the building.

LIGHT MANAGEMENT The light management application relates to the building automation area. This application has two functionalities. It first manages the lighting of the building corridors. The application periodically retrieves the current luminosity level of the building corridors provided by light sensors. If the luminosity level of a corridor is below a given threshold, then when a motion is detected in this corridor, the application switches on the corridor lights. If there is no more motion in the corridor during 10 seconds, it turns off the lights. The second functionality of this application is to manage the lighting of the building halls. The halls have two lighting configurations depending on the building status. Thus, the application is responsible for switching the lighting configuration of the halls when the school opens or closes. The application retrieves the building status from the occupancy schedule already used in the heating control system.

FIRE MANAGEMENT Finally, the fire management application pertains to the emergency management area. This application detects when a fire starts using fire detectors spread throughout the building. When a fire is detected, it turns on the water sprinklers of the fire detection area. Like the anti-intrusion application, it also turns on the building alarms and sends a warning SMS to the building keeper.

The development and simulation of this real-size case study has lead us to develop applications from several pervasive computing areas. Using the DiaSim simulation tool, we were able to test and debug these applications in a simulated platform. Using the same application code, we were able to deploy these applications in our own offices for demonstration purposes, using the same application code. This case study illustrates the usefulness

of a simulation tool when developing pervasive computing applications. A more thorough validation of DiaSim is presented in the next chapter.

7

DIASIM VALIDATION

DiaSim is validated in this chapter. We first evaluate DiaSim with respect to its scalability, usability, and performance. Then, we discuss pragmatic issues involved in testing pervasive computing applications.

7.1 DIASIM EVALUATION

We now conduct an evaluation of DiaSim. To do so, we explore three aspects. We first discuss the *scalability* of our simulation tool. We then study the *usability* of DiaSim, before evaluating its *performance*.

7.1.1 *Scalability*

In our large-scale case study presented in Section 6.2, using simulation allowed us to validate the coordination logic at a large scale, combining 110 entities, 6 stimulus producers, 200 people and 6 applications. Some entities were coordinated and shared by several applications (*e.g.,* `Calendar` and `MotionDetector`). It was thus essential to ensure the usability of these applications by preventing potential conflicts. We also checked that the application behavior met its requirements when the context of deployment or execution changes (*e.g.,* disappearing entities and moving individuals). For example, we improved the Newscast application by making it less sensitive to people that do not stop long enough in front of school LCD screens. We also optimized the air conditioning consumption by combining information about the building occupancy and class schedules.

7.1.2 *Usability*

We have been using DiaSim as part of a course on software architectures for three years. This course includes an 8-hour lab consisting of twenty groups of three master's level students. These students have only followed an introductory course on Java before our course and have basic knowledge of software design and no exposure to the domain of pervasive computing. The goal of our lab is to develop a Newscast application. It

Task	Completion		Avg.
	full	part	time
DiaSpec specification	100%	0%	2h
Implementation	60%	40%	5h
DiaSim simulation	30%	0%	1h

Table 2: Results of a lab involving 60 Master's level students.

requires devices to broadcast messages (*e.g.*, loudspeakers and screens), and devices to identify users (*e.g.*, RFID badge readers). The students have to (1) design the application with DiaSpec, (2) implement it, and (3) simulate it with DiaSim.

The results of this lab are presented in Table 2. Due to the short duration of the lab, last year, only 30% of the students completed their implementation and had enough time to simulate it with DiaSim. The students had to instantiate simulated screens, simulated loudspeakers and simulated badge readers using the DiaSim editor. They also had to add several simulated people to the simulation. Then they had to create a stimulus producer that sends simulated badge detection stimuli when a simulated agent is getting close to a badge reader. We provided them with an online tutorial to help them create their simulation [1]. It is interesting to notice that these students only required on average 1 hour to simulate their application using DiaSim. Though the simulation was simple, it allowed us to get feedback on the usability of DiaSim. In particular, during this lab, simulated people were added programmatically to the simulation. The creation of the simulated people was complicated for the students. Because of that feedback, we modified the DiaSim graphical editor to allow simulated agents to be added graphically to the simulated environment.

This experience demonstrates that students with modest knowledge in software engineering are able to efficiently use DiaSim in a short period of time. However, because the lab is short, the students were only able to achieve a simple simulation. It would be interesting to do another lab focusing especially on the simulation. This would allow to request a more complicated simulation to the students, giving us a more thorough evaluation of the usability of DiaSim.

7.1.3 *Performance*

To study the overhead caused by DiaSim, we evaluated its performance during the engineering school simulation. Our

1. http://diasuite.inria.fr/documentations/tutorial/

goal is to collect measurements when DiaSim is applied to two different simulation workloads: low activity and high activity. The simulation has been executed on a laptop with a CPU Intel Core 2 Duo 2.80 GHz and with 4 GB of RAM. The operating system used by the laptop was Windows XP. The measurements were realized with the JProfiler software.

CPU USAGE. We first evaluated the CPU usage during the simulation. The results of this evaluation are presented in Figure 23. The CPU usage has first been evaluated when the activity is low during the simulation. A low simulation activity is typically during the night or when students are sitting in the classrooms. Then, we evaluated the CPU usage during high activity periods. There is a high activity during the breaks when students are moving in the school. The CPU usage was evaluated with respect to the simulation speed, which ranges from twice as fast as the real time (simulation speed number 1 in Figure 23) to 360 times as fast as the real time (simulation speed number 11 in Figure 23).

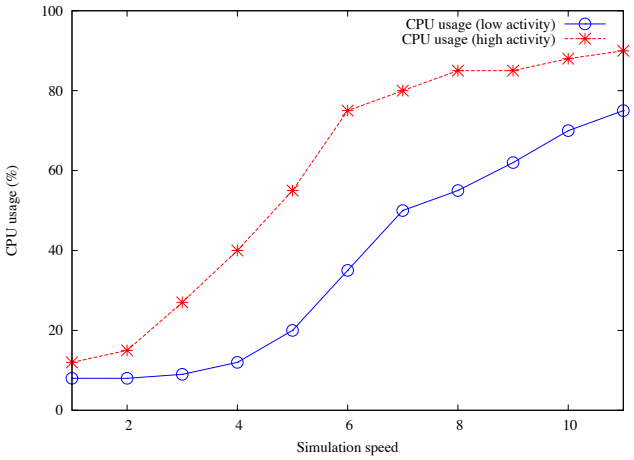

Figure 23: This graph represents the average CPU usage with respect to the simulation speed. The CPU usage has been evaluated during a period of low activity for the simulated agents and during a period of high activity.

This evaluation shows that simulating at a low speed uses less than 20% of CPU. We can also see that simulating at a higher speed requires more CPU. This is due to the graphical rendering that requires to update more often its rendering. To use less CPU, it is possible to disable the graphical rendering and only log the simulation data. It prevents the tester from monitoring the simulation graphically, but it allows him to execute the simulation at a much higher speed while using less CPU.

MEMORY USAGE. To run the simulation, we allocated 1.4 GB of maximum memory to the JVM. The memory is fully used

	Nb. of threads	Percentage
DiaSpec	244	96.06%
DiaSim	10	3.94%

Table 3: Distribution of the threads executed during the ENSEIRB simulation.

during the simulation. This memory is mainly used by Siafu (approximately 1.2 GB to store information concerning the motion of the simulated agents. It uses this information to quickly compute a path to graphically move a simulated agent from one point to another. Thus, simulating fewer people enables to use less memory.

THREAD DISTRIBUTION. We studied the threads used during the simulation. In particular, we studied the thread distribution between DiaSpec and DiaSim. Overall, the vast majority of the threads are related to the DiaSpec runtime, DiaSim accounts for less than 4%. The results of this study are presented in Table 3.

To conclude, it is important to notice that this simulation has been executed on a three-year old laptop, not very powerful compared to today's computers. We would get much better performance results if the simulation were run on a recent computer. Nevertheless, it is possible to execute a large-scale simulation comprising 200 simulated people and 110 simulated entity instances with a modest computer.

7.2 DISCUSSION

We now examine pragmatic issues involved in developing and using a simulated environment. We start by investigating the performance issues involved in running large-scale simulations. Then we discuss the potential pitfalls of our approach.

7.2.1 *Performance*

The simulation of physical spaces may involve lots of entities, accurate simulation models, and rich simulation logic. This situation calls for a scalable simulator.

To support compute-intensive simulation, DiaSpec enables to distribute simulated entities and stimulus producers. This distribution is naturally achieved using DiaSpec because DiaSpec components communicate via a distributed systems technology.

Our implementation of DiaSpec supports several distributed systems technologies including a local software bus, Java RMI, SIP and Web Services. The selection of this distributed technology is done at deployment time and does not affect DiaSpec component implementation. When the simulation back-end used by DiaSim is Java RMI, the workload can be distributed over several different hosts, enabling numerous entities and stimulus producers to be introduced. A distributed technology also makes it possible to perform hybrid simulation by integrating distributed, real entities.

7.2.2 *Pitfalls*

A simulation consists of tested applications and the simulated environment. The output of the simulated environment is the input of the tested applications and vice versa. The complexity of the simulated environment depends on the characteristics of the real environment and how accurately it needs to be modeled. These issues go beyond the scope of our generated simulation support that is aimed to facilitate the programming of the simulated environment. Producing faithful stimuli and defining meaningful simulation logic are left to the developer.

Specifically, the values generated by a stimulus producer need to be faithful to some simulation model. The simulation model must provide an accuracy that matches the granularity of the situations to be tested. To define a stimulus producer, one option is to replay data logged from entity data sources, whether or not verbatim. Another option is to define a stimulus producer using some domain-specific modeling function. Issues about the correctness of the stimulus producer arise when either the logged data are transformed or a domain-specific modeling function is introduced. Beyond stimulus producers, emulated entity actions may have an effect on the simulated environment (*e.g.*, a light impacts the luminosity). As a result, the stimulus producers need to subscribe to all entity actions that may have an effect on the values they generate.

To illustrate these issues consider the sun luminosity. It can simply be defined by a mathematical function. However, its impact on a building is difficult to model as it depends on the number, size and location of windows, and the building structure. Our approach does not help in defining an accurate model of this situation; this is left to the simulation developer that must take into account the simulation requirements.

Another source of inaccuracy may be created by the operations that merge stimulus intensities produced by the same region of

the physical space. For example, consider the luminosity in a hall coming from the luminosity of the surrounding rooms. These luminosity intensities are sent to the luminosity producer of the hall, which merges them and passes the new intensity to the hall light sensors. This merging operation is also user-defined; to be meaningful its definition needs to rely on domain-specific knowledge.

As one can see, taking into account the simulation requirements and developing stimulus producers in Java can be laborious. It often requires to encode in Java mathematical formulas describing the stimulus producer evolution. To reduce this complexity, we have worked on easing simulation of natural phenomena [12]. To do so, we leverage Acumen [76], a DSL for describing differential equations. The differential equations defined with Acumen describe physical phenomena. With Acumen, we use off-the-shelf physical environment models and formulas that are available in textbooks and the research literature. Their correctness is extensively documented and well established. Leveraging a physical environment modeling language such as Acumen allows us to both reduce the stimulus producer implementation complexity and ensure the correctness of our stimulus producers. This work is described in the following chapter.

Entities are emulated so that applications interact with them without code modification. To be faithful, an emulated entity should have an observable behavior that is equivalent to its real counterpart. To do so, the data source of an emulated entity can be programmed such that, for a given input, it produces the same output as its real counterpart.

8

PHYSICALLY-ACCURATE TESTING

Testing pervasive computing applications is a crucial tool for eliminating poor designs, and developing a degree of confidence in promising designs. But testing pervasive computing applications after deployment can be slow and prohibitively expensive. Achieving the testing virtually by using simulation helps solving these two issues. The effectiveness of this approach, however, depends heavily on the accuracy with which we model both the application and the physical environment interacting with the application. Furthermore, building accurate simulation codes, especially for physical environment, can be labor intensive and can slow down the whole testing process of pervasive computing applications.

Three technical challenges must be overcome in order to enable an effective physically-accurate testing of pervasive computing applications. The first is to accurately capture the distributed and networked nature of the pervasive computing application. The second is to accurately model the physical environment. The third is to automatically map such models directly to executable simulation codes.

Existing approaches only cope at most with one of the three challenges raised by the physically-accurate testing of pervasive computing applications. For instance, several projects simulated devices using MATLAB/Simulink [60]. These projects focus on the fine-grained modeling of these devices. They provide libraries of digital components that can be used to model devices. However, they do not attempt to use analytically sound models of the physical environment surrounding such devices. COMSOL allows to accurately simulate the surrounding physical environment. For instance, it provides a heat transfer module and an acoustics one. However, these simulations are based on the Finite Element Method (FEM) and are prohibitively expensive for modeling the physical environment of a whole building for example. Other tools allow faster simulation of the physical properties. Modelica is one of these tools. Modelica is an equation-oriented modeling language. The main draw back is that modeling systems in Modelica that combine discrete and continuous behaviors can be somewhat challenging.

We address the three technical challenges for achieving a physically-accurate testing of pervasive computing applications.

We address the first challenge by modeling the applications with DiaSpec, which allows modeling distributed and networked pervasive computing applications. The physical environment aspects are modeled explicitly in Acumen [76], a domain-specific language with specialized support for describing continuous systems. The complete models, containing both Acumen and DiaSpec components are mapped to executable codes. This is achieved by combining Acumen's simulation capability and DiaSim. Combining Acumen to DiaSim allows to address the two remaining challenges.

In this chapter, we present our physically-accurate testing approach. We first present how the physical environment is modeled using Acumen. We then explain how DiaSim and Acumen were combined and executed in a same simulation. We then illustrate the interest of a physically-accurate testing with a virtual experiment of our heating control system. This experiment shows how a physically-accurate testing can be used to analyze different heating strategies of our heating control system. Finally, we discuss the limitations of our approach that still need to be overcome.

8.1 MODELING THE PHYSICAL ENVIRONMENT

The first step for a physically-accurate testing is to accurately capture the properties of the physical environment. In our case study, our heating control system regulates the temperature in a building. A building is a multi-physics system involving multiple interrelated physical characteristics. Physicists have already defined these phenomena with mathematical definitions (*e.g.*, with ordinary/partial differential equations). These analytical descriptions are an ideal material to reuse in order to capture the physical properties of a building. In this section, we first explain the approach that we adopt to modeling heat transfer and temperature change in a building. We then show how the differential equations that capture this model are expressed in Acumen.

8.1.1 *Modeling Temperature and Heat Transfer in a Building*

Human comfort and safety are highly sensitive to the temperature of the surrounding air. As a result, it is critically important to accurately model the factors that impact temperature, including appliances that can be used to control it, as well as the processes by which the temperature of the air in a given room is changed.

We present the models used in our study as a series of successive refinements of a basic model. All models are compartment models, in that they treat each room as one state variable. The basic model, as well as the refinements which include additional terms, are all differential equations.

8.1.1.1 *Heat Transfer.*

Heat transfer is the rate at which heat moves through a medium or from one medium to another, and is a topic studied extensively in thermodynamics (*e.g.,* [41]). Heat transfer between two media is linearly proportional to the difference in temperature between the two media. In addition, it is also affected by the thermal resistance of the boundary between the two media. For simplicity, we assign each room in a building one temperature value. Reasonable values for the thermal resistance of building walls, windows and doors can be determined using a reference book in the heating and cooling domain [41].

Let us assume that all rooms are numbered. We will use the subscripts i and k to refer to room numbers. Let $\mathtt{Neighbors}(i)$ be the set of numbers representing the rooms neighboring room i. Let T_i denote the temperature of room i.

To help introduce the reader to the notation and the equations that define heat transfer in a building, we begin by assuming that the only factor affecting temperature in a particular room is heat from neighboring rooms. To express even this simple process, we need some additional notation. In particular, we will also use the following convention:

$\frac{dT_i}{dt}$	Rate of temperature change in room i ($^\circ C.h^{-1}$),
C_i	Thermal capacitance of room i ($J.^\circ C^{-1}$),
R_{ik}	Thermal resistance of the boundary between rooms i and k ($^\circ C.h.J^{-1}$). It takes into account the heterogenous elements of this boundary (*e.g.,* walls, windows, doors).

The equation constraining the rate of change for each and every room i is given by the equation:

$$\frac{dT_i}{dt} = \frac{1}{C_i} \sum_{k \in \mathtt{Neighbors}(i)} \frac{T_k - T_i}{R_{ik}} \tag{8.1}$$

Because this equation is instantiated for each room, the whole building is modeled by a set of such equations.

8.1.1.2 *Air-Conditioning Unit.*

We now consider adding air-conditioning (AC) units to our model. An AC unit consists of a heater and a cooler. We need only to introduce four additional parameters:

$B_{h(i)}$	The heater in room i is active (1 if present and active, 0 o.w.),
P_i	Heater power (W),
$B_{c(i)}$	The cooler in room i is active (1 if present and active, 0 o.w.),
Q_i	Cooler power (W),

The equation above only needs to be extended as follows:

$$\begin{aligned}\frac{dT_i}{dt} =& \frac{1}{C_i} \sum_{k \in Neighbors(i)} \frac{T_k - T_i}{R_{ik}} \\ &+ \frac{1}{C_i} * (B_{h(i)} * P_i - B_{c(i)} * Q_i)\end{aligned} \tag{8.2}$$

8.1.1.3 *Occupants.*

Occupants can be modeled as heat sources. The set of occupants of room i is denoted $Occupants(i)$. We only need one additional parameter to incorporate this aspect of building:

H_j	Heat dissipation of occupant number j (W),

Thus, the final equation can be expressed as:

$$\begin{aligned}\frac{dT_i}{dt} =& \frac{1}{C_i} \sum_{k \in Neighbors(i)} \frac{T_k - T_i}{R_{ik}} \\ &+ \frac{1}{C_i} * (B_{h(i)} * P_i - B_{c(i)} * Q_i) \\ &+ \frac{1}{C_i} * \sum_{j \in Occupants(i)} H_j\end{aligned} \tag{8.3}$$

Other heat sources, such as equipment, appliances, and lights, were neglected for simplicity but will be included at a later stage. Now, we are ready to present the actual Acumen code used in the experiments we report on in the rest of this chapter.

8.1.2 *The Heat Model in Acumen*

By design, the Acumen modeling language enables direct specification and simulation of continuous and discrete systems. Our

virtual testing framework only uses its continuous system modeling and simulation capability, and not its support for modeling discrete behaviors.

Figure 24 presents the Acumen specification of the temperature defined in Equation 8.3. The `continuous` section in Figure 24 specifies the temperature rate of change for each room of the building. Equations in Acumen can refer to derivatives of variables. For example, `T'` refers to the first derivative of `T`. Finally, the boundary conditions subsection allows one to define the initial state of the physical environment. This initial state can be easily changed by setting new boundary conditions. As can be noticed, defining physical characteristics in Acumen is straightforward and has a direct correspondance to equational definitions. Thus, Acumen leverages standard mathematical notation used to define physical phenomena.

```
(* Building topology, Room 0 corresponds to the outside *)
building=((0),(0,2),(0,1,3),(0,2));

(* Temperature in each building room, T0 is the outside
    temperature *)
T=(T0,T1,T2,T3);

(* Other variable definitions *)
...
continuous
  foreach room in length(building) begin
    T'[room] = 1/C[room] *
      ((sum n < length(building[room])  in
          ((T[building[room][n]]-T[room]) / R_th[room][n]))
        + Bh[room]*P[room]-Bc[room]*Q[room]
        + (sum p < length(occupants[room])  in
            H[occupants[room][p]]));
  end

 boundary conditions
   T0 with T0(0) = 10;
  (* We define the boundary conditions for all continuous
      variables *)
   ...
```

Figure 24: Temperature specification in Acumen.

8.2 MAPPING THE MODELS INTO EXECUTABLE SIMULATION CODES

The final technical challenge to enabling effective physically-accurate testing of pervasive computing application is to automatically map the models of both physical environment and application to executable simulation codes. This section explains

how this mapping is done, and details how the simulations of the physical models and DiaSpec models interact.

8.2.1 *Execution of Physical Models*

Physical models are directly mapped to executable code. The process of mapping physical models to executable code is specific to the modeling tool considered (*e.g.,* Acumen [76]). Continuous variables in a physical model are discretized with respect to a user-defined step size. The user needs to properly set the step size value so that an acceptable level of accuracy is obtained in the simulation.

8.2.2 *Physically-Accurate Simulation*

The physical model and DiaSim simulations interact in two ways. First, simulated sensors retrieve their sensed information from the physical model. Second, simulated actuators may modify the state of variables in the physical model. These two interactions are achieved using a socket-based Java API. [1] While editing the simulated pervasive computing application in the DiaSim editor, the user specifies the physical model variables sensed by each simulated sensor. He also specifies which variable is modified by an actuator action and how it is modified. Finally, we generate Java code for executing simulated devices and interfacing these devices to the physical model.

8.3 MONITORING PHYSICALLY-ACCURATE SIMULATION

To help the tester to determine if an application behaves correctly, DiaSim provides monitoring and rendering functionalities. In the case of virtual testing, we also provide tools for analyzing the results after the completion of the physically-accurate testing. During a simulation, we log the values of the continuous and discrete variables. This allows to save the physical environmnent state and the simulated device states at each simulation iteration. At the end of a simulation, all logged variables are automatically plotted. This plot can serve as a basis for analyzing events that occured during the simulation. If the user is interested in a more specific analysis (*e.g.,* energy consumption of the active devices), it is easy to create dedicated plots with tools such as Gnuplot [2]. For example, we created Gnuplot scripts for reading

1. Thanks to Cherif Salama for providing this API.
2. http://www.gnuplot.info/

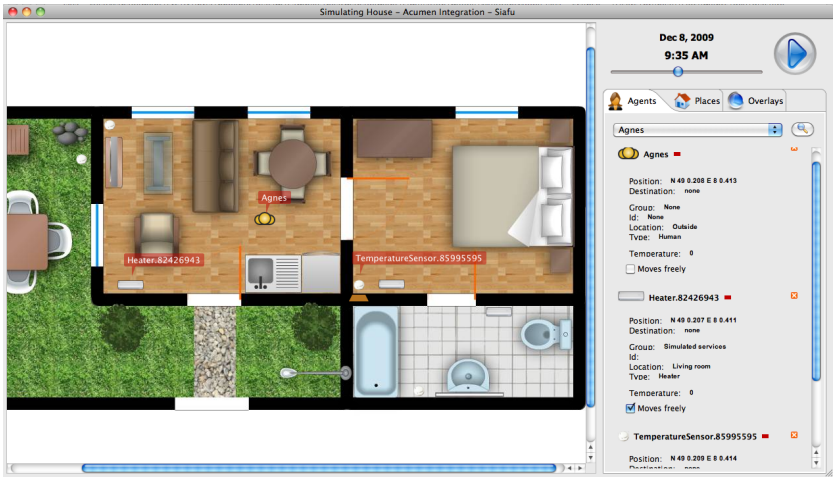

Figure 25: 2D graphical rendering of a virtual house.

the logged variables and displaying the energy use and comfort level achieved in Section 8.4.2.

8.4 A VIRTUAL EXPERIMENT

This section presents the analyses that enable our physically-accurate testing framework. Pervasive computing applications can first be evaluated using *algorithmic variations*. Indeed, multiple algorithms for the same application can be compared. Virtually testing these algorithms can help us find the most efficient one with respect to the functionalities of the pervasive computing application. This variation is facilitated by the DiaSpec approach, as it confines the algorithmic variations to only some context components. A second variation allowed is *structural variation*. Multiple configurations of sensors and actuators can be tried to choose the most efficient one. This variation is also facilitated by DiaSpec, as the application logic does not need to be modified when structural variations are applied.

To illustrate both types of variation, we evaluate our heating control system deployed in a house. Figure 25 illustrates the simulation of this system displayed in the DiaSim simulation renderer. We apply algorithmic and structural variations to this system for illustrating the usefulness of our physically-accurate testing framework.

8.4.1 *Global vs. Local Temperature Management*

Multiple Regulating and Standards organizations define comfortable ranges of temperature depending on the indoor humidity and the outside temperature. For instance, ASHRAE defines the

comfortable humidity and temperature ranges in its "Thermal Environmental Conditions for Human Occupancy" standard. We use these values for evaluating our HVAC control algorithms.

Originally, to keep the temperature in the comfort zone, HVAC systems had a single thermostat and regulated the temperature everywhere using this single thermostat. We call this strategy *global temperature management*. Global temperature management is obviously not optimal, because the temperature can be different in each room. To achieve finer grained, and more efficient regulation, EnergyStar recommends managing the temperature locally [25]. In this case, the building is divided in areas, each area with its own thermostat. We call this strategy *local temperature management*. To illustrate virtual testing, we set up an experiment to evaluate the effect of applying this recommendation to the heating control system of a 3-room house. We then observe the comfort and the energy use differences between these two temperature management policies.

8.4.2 *Heating Control System Evaluation*

Both global and local regulations use the same regulation logic. The only difference is the scope of this regulation. We name T_{c-min} and T_{c-max}, respectively the lowest comfortable temperature and the highest comfortable temperature. Our regulator needs to keep the temperature in a range $[T_{min}, T_{max}]$. If the temperature is below T_{min}, the HVAC system ventilates hot air. If the temperature is above T_{max}, the hot air stops being ventilated. Obviously, the logic of our regulator is very simple and could be refined. However, it is enough for illustrating the variations we apply for testing this regulator.

In this virtual experiment, we use the temperature model presented in Section 8.1. In this model, our virtually tested HVAC system has a heating power of 1500 Watts and a cooling power of 600 Watts. The calculated thermal resistance coefficients depend on the windows, doors and walls that compose these boundaries.

8.4.2.1 *Algorithmic Variation.*

In this section, we test several algorithms for our temperature regulator and choose the most efficient one with respect to its energy use. T_{min} and T_{max} are defined in Equation 8.4.

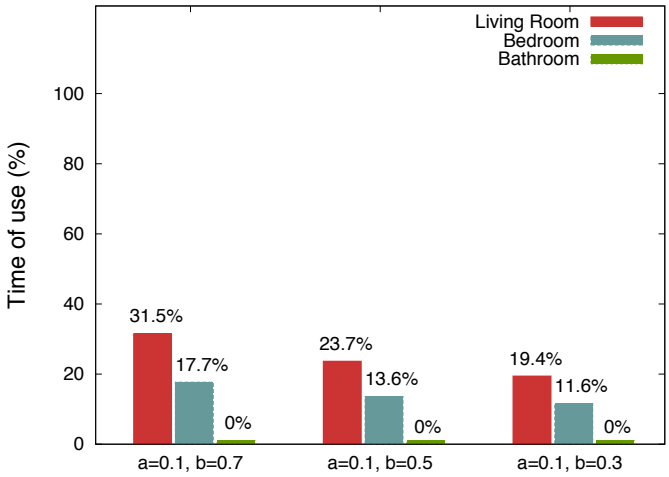

Figure 26: Comparison of different algorithms for regulating the temperature in terms of energy use.

$$\begin{aligned} T_{min} &= T_{c-min} + a * (T_{c-max} - T_{c-min}) \\ T_{max} &= T_{c-min} + b * (T_{c-max} - T_{c-min}) \\ & a, b \in [0, 1] \text{ and } a < b \end{aligned} \tag{8.4}$$

We test our HVAC system with different values of T_{min} and T_{max} over a period of one month. We decrease these two values as long as comfort is provided 100% of the time. We choose January because heating is critical during this month. The chosen outside temperatures correspond to Bordeaux average temperatures in January. We test three different algorithms. The percentage of time when the hot air is ventilated is presented in Figure 26. We see that these algorithmic variations bring differences in terms of heater time of use. A low range of temperature results in less heating time. Since the heaters have the same power, the differences in the time of use of these heaters allow to evaluate the energy consumption gain. The third algorithm in Figure 26 allows a 38% gain of energy use compared to the first one.

8.4.2.2 *Structural Variation.*

We also test two different device deployments for comparing global and local temperature regulation. The global configuration consists of one temperature sensor in the living-room and one heater in each room. The local configuration consists of one temperature sensor and one heater in each room. We use the most efficient algorithm from the previous section. We test our HVAC system over the month of January. The comfort provided by these two configurations is presented in Figure 27. Local temperature

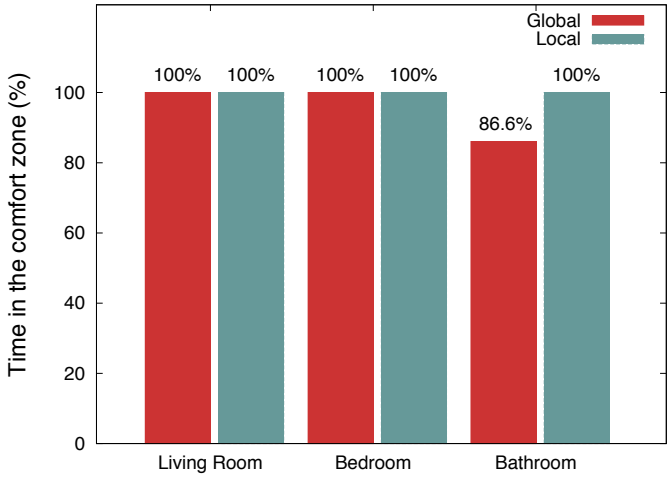

Figure 27: Comparison of the comfort provided by global and local temperature management in the house.

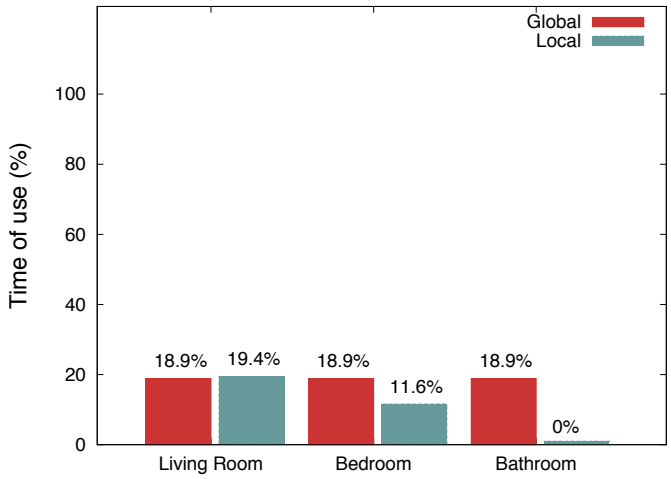

Figure 28: Comparison of the time of heating required by global and local temperature management in the house.

regulation provides perfect comfort to the three rooms of the house. In comparison, global temperature regulation is not as comfortable for the occupants. The bathroom temperature is comfortable for only 86% of the time. The percentage of time when hot air is ventilated for the two types of regulation is presented in Figure 28. We can see that global temperature management ventilates more hot air than local temperature management. Our physically-accurate testing framework allowed us to see that following the EnergyStar recommandation enables to get a better comfort with less energy use from the HVAC system.

8.5 DISCUSSION

In this section, we discuss two important issues involved when using a simulated environment, namely *accuracy* and *validity*.

8.5.1 *Accuracy*

The user needs to carefully choose the accuracy of his simulation. The accuracy of the simulation depends on two parameters: physical environment model accuracy and size of the time discretization step. The temperature model we present in this paper considers that each room has a single state. A smaller discretization of the space would allow the simulation of the physical environment to be more precise. However, a smaller space discretization requires more computation.

Likewise, the size of the time discretization step impacts the virtual testing accuracy. In our evaluation, we choose a discretization step of one minute. A one minute accuracy is enough for evaluating an HVAC system over a month. However, this criteria needs to be carefully chosen depending on the tested smart building application.

8.5.2 *Experiment validity*

As described in this chapter, our simulation approach enables to test and compare different implementations and/or configurations of pervasive computing applications. However, we still need to validate our simulation framework to ensure that the output of our simulations are valid. There are two approaches to validate our simulator. The first approach is to compare a simulated execution and a real execution of the same pervasive computing application. To validate our simulation framework, the simulation output needs to be sufficiently close to the reality.

However, it is a complicated task to compare the simulation of a pervasive computing application with the reality. First, we would need to buy and deploy all the necessary devices, which is time-consuming and expensive. Then, we would need to log data of this deployment for a long period of time. For instance, we would need to log data during at least one month if we wanted to validate the experiment we present in this chapter.

A faster and simpler way to validate our simulation framework is to compare our simulation framework with an already validated simulator. If we successfully show that, for a given initial state of the physical environment, the simulation results are similar between the two executions, then this demonstrates that the experiment realized in our simulator is valid. This is the strategy that we plan on following in a future work to validate our simulation framework. We plan on implementing our temperature regulation experiment in the EnergyPlus software [73]. EnergyPlus allows to simulate the energy consumption of a building. Among other physical properties, EnergyPlus models heating, cooling and ventilation. This simulator has been thoroughly tested and validated, the validation results are available online [3]. In this future work, we will interface our heating control system with EnergyPlus and compare the EnergyPlus output with the results presented in this chapter.

Though our simulation framework has been validated yet, it gives the tester an idea of how his application behaves. This first approximation is enough for allowing the tester to eliminate some unefficient control algorithms, such as the global regulation algorithm presented in this chapter. More fine-grained optimizations of the application would require a validation of our simulation tool. For instance, defining the most efficient heater location in the house requires a precise simulation.

3. http://apps1.eere.energy.gov/buildings/energyplus/energyplus_testing.cfm

9

GENERALIZATION OF OUR SIMULATION APPROACH

In this document, our simulation approach focuses on simulating pervasive computing applications. However, the wider the scope of DiaSpec, the more simulation aspects need to be addressed. In this chapter, we first present extensions to DiaSpec that can be simulated with our simulation approach. Then, we present how our simulation approach can be applied to the avionics application domain.

9.1 AN INTEGRATED APPROACH TO SIMULATION

Recent works in our research group have expanded the DiaSpec language in two directions: allowing end users to visually develop pervasive computing applications, and adding non-functional concerns to a DiaSpec description.

TARGETING END USERS Users from the home automation and assisted living domains have shown interest in developing pervasive computing application themselves. However, our methodology requires an area expert to define the entity taxonomy. It also requires an application architect and Java developers. To address these concerns, Drey *et al.* proposed Pantagruel [23], a visual language allowing end users to develop pervasive computing applications. Pantagruel directly generates DiaSpec specifications and implementations. Our DiaSim simulation tool has been useful to this end-user programming approach by providing a visual way to test applications [22].

COVERING NON-FUNCTIONAL CONCERNS Recent works in our research group have expanded the scope of DiaSpec to cover non-functional concerns, extending the DiaSpec language and its compiler. These extensions include (1) handling access conflicts to resources of a pervasive computing system [37], (2) modeling entity failures at the declaration level, enforcing their treatment at the programming level [49], and (3) declaring performance constraints, ensuring them at compile time and run time [32]. Each of these non-functional concerns expand the opportunities of simulation. In fact, we successfully applied our simulation approach to the avionics domain [13]. Specifically, we developed

an aircraft guidance system, using stimulus producers for simulating entity failures. We are planning to extend DiaSim with the simulation of non-functional concerns that are now available in DiaSpec. We present the application of the simulation approach to the avionics domain in the following section.

9.2 REALISTIC SIMULATION OF AN AVIONICS SYSTEM

So far, we have presented a simulation approach for pervasive computing applications. However, this simulation approach is not specific to the pervasive computing domain. We demonstrate in this section that our simulation approach is generic enough to simulate an avionics system: a *flight guidance application* [13].

9.2.1 *Flight Guidance Application*

We applied our development methodology to a flight guidance application. We now present a description of this application and briefly introduce its DiaSpec design.

9.2.1.1 *Description*

The flight guidance application is in charge of the plane navigation and is under the supervision of the pilot. For example, the pilot can directly specify flight parameters (e.g., the altitude) or define a flight plan. Each parameter is handled by a specific navigation mode (e.g., altitude mode, heading mode). Once a mode is selected by the pilot, the flight guidance application is in charge of operating the ailerons and the elevators to reach the target position. For example, if the pilot specifies a heading to follow, the application compares it to the current heading, sensed by devices such as the Inertial Reference Unit, and maneuvers the ailerons accordingly.

9.2.1.2 *DiaSpec Design*

We now briefly describe the taxonomy of entities and the architecture of our flight guidance application. Figure 29 presents the design fragment of our application related to the heading mode. The heading mode allows the aircraft to follow either a heading defined by the pilot, or the heading to reach the next waypoint in the flight plan.

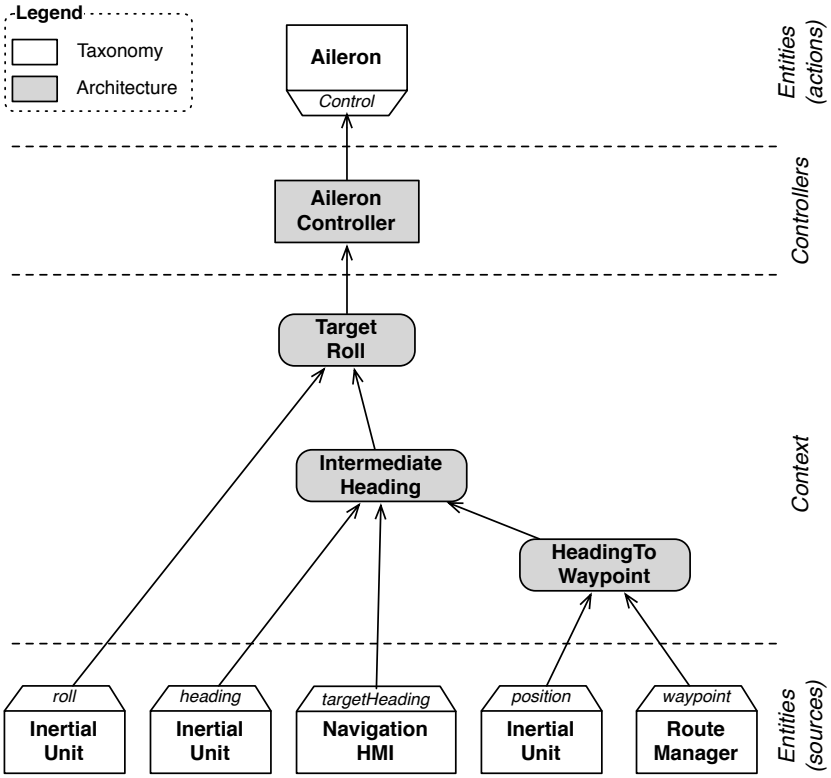

Figure 29: Specification of the flight guidance application application.

TAXONOMY We first identify the entities that are required to control the heading. The aircraft heading is provided by Inertial Reference Units (IRUs). These units encapsulate accelerometers, gyroscopes, and GPS sensors, and provide navigation data. To allow the pilot to set a heading, we define a user-interaction entity, namely Human-Machine Interface (HMI). Finally, controlling the plane heading requires to act on the plane ailerons. These entities are at the top and bottom of Figure 29. The `InertialUnit` entity senses the position, the roll and the heading of the plane from the environment. The `NavigationHMI` entity abstracts over the pilot interaction and directly provides the target heading. The `RouteManager` entity provides the next waypoint information. Finally, the `Aileron` entity provides the `Control` interface to the application.

ARCHITECTURE From bottom to top in Figure 29, the architecture can be summarized as follows. The `HeadingToWaypoint` context component computes a target heading to reach the next waypoint provided by the route manager. The `IntermediateHeading` context component abstracts over the computation of the target heading. Indeed, it can be computed either from `targetHeading` directly provided by `NavigationHMI` or from the target heading computed by `HeadingToWaypoint`. Given this heading and the

current plane roll (*i.e.*, its rotation on the longitudinal axis), the `TargetRoll` context component computes a target roll. This target roll is used by `AileronController` to control the ailerons and reach the target heading.

9.2.2 *Simulation of the Flight Guidance System*

The simulation is required in avionics as applications are confronted to a wide range of potential scenarios. For example, it is required to verify the behavior of the application in specific environmental conditions, which are difficult to create (e. g. extreme flight conditions). To be able to simulate our flight guidance application, we generalized our simulation approach and provided a testing support that relies on a flight simulator, namely FlightGear [54], to simulate the external environment. We chose FlightGear as it is a widely used, realistic flight simulator in the avionics domain.

9.2.2.1 *Simulation Model in the Avionics Domain*

To simulate SCC applications in the avionics domain, we applied the same simulation model as presented in Figure 13. The simulation model of an avionics application is presented in Figure 30. We used FlightGear for simulating the aircraft and its external environment. Thus, FlightGear provides simulated stimuli to our simulated entities. For instance, the roll angle, GPS position, and heading of the simulated aircraft are provided by FlightGear. Our simulation model also enables the actuators to impact FlightGear. Thus, the ailerons of the aircraft, simulated in FlightGear, are impacted when the `Aileron` entity is controled by our flight guidance application. This allows our application to control the simulated aircraft in FlightGear.

9.2.2.2 *Interfacing FlightGear*

FlightGear allows to read and write its current state using socket connections. We used this FlightGear capability and developed a Java library to interface with FlightGear. The testers can easily implement simulated versions of entities using this library. Figure 31 presents an extract of the implementation of a simulated `InertialUnit` entity.

The `SimulatedInertialUnit` entity is implemented by inheriting the `AbstractInertialUnit` class provided by the programming framework. To interact with the simulated environment, the entity implements the `SimulatorListener` interface. This interface defines a method named `simulationUpdated`, which is

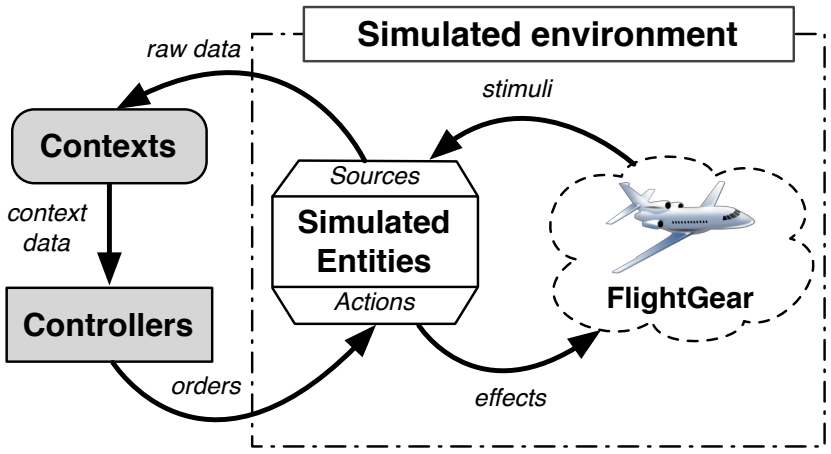

Figure 30: Simulation model of an avionics application.

```
public class SimulatedInertialUnit extends AbstractInertialUnit
               implements SimulatorListener {

  public SimulatedInertialUnit(FGModel model) {
    super();
    model.addListener(this);
  }

  public void simulationUpdated(FGModel model) {
    publishPosition(model.getInertialPosition());
  }

  [ ... ]
}
```

Figure 31: Extract of the implementation of a simulated `InertialUnit`.

called periodically by the simulation library. The `model` parameter allows to read/write the current state of the FlightGear simulator. In Figure 31, the position of the plane is published by calling the `publishPosition` method of the `AbstractInertialUnit` class.

9.2.2.3 *Execution of the Simulation*

Once the simulated entities are implemented, the flight guidance application is tested by controlling a simulated plane within FlightGear. Figure 32 presents a screenshot of our testing environment. In the main window, the FlightGear simulator allows to control and visualize the simulated plane. In the top-left corner, the autopilot interface allows testers to select a navigation mode. In this case, the "Route Manager" mode is selected to follow

the flight plan defined via the map displayed in the bottom-left corner.

We also leveraged FlightGear capabilities to simulate instrument failures. Thus, the window in the top-right corner in Figure 32 allows to cause IRU failures. Finally, the window in the bottom-right of the screenshot logs the application execution.

A video illustrating the development and simulation of this flight guidance application is available online [1].

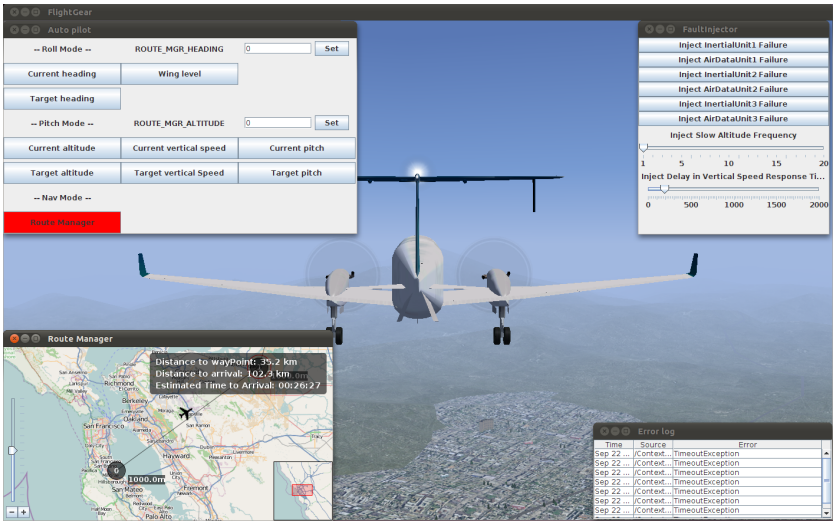

Figure 32: Screenshot of a simulated flight.

9.2.3 *Conclusion*

In this section, we showed how to generalize our simulation approach to the avionics domain. In fact, our simulation approach may be applied to any application domain, as long as the DiaSpec approach is used to design the application. When an application is developed using DiaSpec, the tester has two options to simulate the application. He can first develop his own stimulus producers that fit the application domain he needs to target. However, developing stimulus producers of the physical aspects of a simulated environment is a very complex task. Moreover, if the tester is not an expert of the domain, the stimulus producers are unlikely to be physically accurate. The second approach is to use an existing simulator (*e.g.*, FlightGear in the avionics domain) and implement simulated entities that interact with this simulator. The drawback of this option is that interfacing these domain-specific simulators may be complicated to implement. However, this option allows to leverage the simulation capabilities of often very powerful domain-specific simulators.

1. `http://diasuite.inria.fr/avionics/51`

Part IV

CONCLUSION

10

RELATED WORK

In this chapter, we study other existing approaches for developing and testing pervasive computing applications. We first present the existing development methodologies that provide support for the entire development cycle, from the design stage to the testing stage. Then, we present the development approaches that target particular stages of the development cycle.

10.1 MODEL-DRIVEN ENGINEERING

Model-Driven Engineering (MDE) uses models and model transformations to specify software architectures and generate implementations [63]. The goal of these MDE approaches is to raise the abstraction level in program specifications and generate a working implementation from such a specification. UML 2.0 (Unified Modeling Language) has been widely accepted as an architecture modeling notation [8] and as a second-generation ADL [48]. Various development environments, relying on UML and MDE, have been proposed (*e.g.,* Enterprise Architect [26]). These development environments cover the complete development life-cycle. However, they do not target the specific features of pervasive computing, leaving the customization work to the architects and the developers.

PervML [64] customizes the MDE approach with respect to the domain of pervasive computing by proposing a conceptual framework for context-aware applications. This conceptual framework relies on UML diagrams to model pervasive computing concerns. For example, services are modeled with class, sequence, and state transition diagrams, while locations are modeled with package diagrams. Even though the conceptual framework proposed by PervML is domain specific, it relies on generic notations and generic tools, incuring an overhead for designers.

In contrast, DiaSpec designers only manipulate domain-specific concepts and notations (*e.g.,* entities, context and controller components), facilitating the design phase. PervML, along with most MDE-based approaches, require designers to directly manipulate OCL and UML diagrams. As reported in the literature, this manipulation becomes "enormous, ambiguous, and unwieldy" [28, 55, 69]. In practice, these approaches demand an in-depth expertise in MDE technologies.

From UML diagrams, PervML provides a dedicated suite of tools to generate a complete implementation. In contrast, DiaGen provides a dedicated programming support to the developer, but it does not automate the implementation stage. Moreover, a strong advantage of using UML diagrams is that developers' knowledge and existing tools are leveraged, such as the Eclipse Graphical Modeling Framework (GMF).

Finally, PervML only offers rudimentary testing support, based on device simulation. Contrary to our testing support, theirs does not allow to test PervML applications in a simulated physical environment, nor does it allow to have simulated people interact with the applications.

10.2 ARCHITECTURE DESCRIPTION LANGUAGES

Architecture Description Languages (ADLs) are used to make explicit the design of an application. Most ADLs are dedicated to analyzing architectures; they provide little or no implementation support. Archface [72] is the most recent instance of this line of work. It is both a general-purpose ADL and a programming-level interface. It proposes an interface between design and code. However, the design support provided by Archface is generic. Furthermore, Archface requires the software architect to have some knowledge about the implementation layer to be able to express the interface part of a design.

In contrast, our approach is domain specific and thus allows domain experts to design their architecture without implementation knowledge. The design is then used to generate dedicated programming support for the developer. For example, DiaGen generates dedicated programming support to discover entities based on the taxonomy definition. In Archface, a design is directly mapped into programming-level interfaces, ensuring the conformance between the design and the implementation. However, unlike our approach, Archface does not provide dedicated programming support. The testing stage is not covered by this approach.

10.3 CONTEXT MANAGEMENT MIDDLEWARES

Numerous middlewares have been proposed to support the implementation of pervasive computing applications. Schmidt *et al.* [62], Chen and Kotz [17], and Dey *et al.* [21] have proposed middleware layers to specifically acquire and process context information from sensors. Henricksen *et al.* take this approach

one step further by introducing a language to model the computation of context information [35, 46]. However, none of these middlewares provide tool support for the design phase, they only provide design guidelines.

Although, context management middlewares provide programming support for acquiring and processing context information from sensors, they do not address the other activities pertaining to a pervasive computing application (*e.g.*, device actuation). The other development stages are not covered by these middlewares.

10.4 PROGRAMMING FRAMEWORKS

The programming framework approach has been applied to the domain of pervasive computing to facilitate the development of applications by raising the level of abstraction. A representative example is Olympus [58]. Olympus offers limited support for the design stage: it mainly consists of guidelines related to the concept of Active Space.

An Active Space represents a physical space enriched with sensors and actuators. Virtual entities of an Active Space can be described programmatically using high-level programming interfaces, allowing the developer to focus on the application logic. However, the programming support is not dedicated to a specific description of an Active Space. Thus, the application logic is implemented using generic datatypes, making the implementation error-prone. In comparison, with our approach, the developer is provided datatypes that are dedicated to the application to be implemented.

Player is a framework and a middleware in the robotics community, widely recognized as a standard for robot programming [18]. It has successfully been applied to pervasive computing applications in the kitchen environment [40]. Approaches such as Player rely on a fixed programming framework, requiring it to cover as much of the target domain as possible. This situation results in large APIs, overwhelming the developer and requiring boilerplate code to customize the programming framework to an application area. In contrast, a DiaGen-generated programming framework specifically targets one application, limiting APIs to methods of interest to the developer. In principle, our code generator DiaGen could target these middlewares and programming frameworks thus leveraging existing work.

10.5 SIMULATORS

In this section, we study the existing simulation tools for the pervasive computing domain. We also study simulators from two simulation fields related to the pervasive computing domain: context simulators and networked entities simulators. Finally, we discuss existing approaches for achieving physically-accurate simulation.

Pervasive Computing Simulators

Few simulators are dedicated to the testing of pervasive computing applications. Stage and Gazebo are simulators dedicated to the Player programming framework and have been used to simulate a sensor-enriched kitchen [40]. Player is a programming framework and a middleware created in the robotics domain and widely recognized as a standard for robot programming [18]. Player allows to specify interfaces that define how to interact with robotic sensors, actuators and algorithms. However, this design support is very limited as it does not cover the design of other application components. For instance, it does not allow to design the controllers that coordinate robotic devices.

The Player programming support enables to develop a wide range of robotic applications. However, this programming support only targets the robotic area. In contrast, our approach can be used in any domain in which the SCC architectural pattern applies (*e.g.,* avionic, robotic, pervasive computing). Player applications can be simulated in a 2D graphical environment using Stage, or in 3D using Gazebo. However, both simulators only target the simulation of mobile robot. Moreover, they have to be manually specialized for every new application area. In contrast, DiaSim relies on the DiaSpec descriptions to automatically customize the simulation tools (*i.e.,* the scenario editor and the simulation renderer).

Other pervasive computing simulators include Ubiwise [4] and Tatus [53] that are built upon 3D first-person game-rendering engines, respectively Quake III Arena and Half-Life. They allow the user to have a focused experience of a simulated environment. However, these simulators are difficult to extend: the game-rendering engine has to be modified to add new sensors and actuators, or to simulate arbitrary context data.

The Lancaster simulator enables deterministic testing conditions and emulation to test location-based applications [50]. However, libraries of actuators and sensors are not provided and the development of new types of sensors and actuators is not

supported. The PiCSE simulator addresses the problem of extensibility by providing generic libraries to create sensors and actuators [59]. However all these approaches do not propose an emulation framework to incrementally integrate real entities in a simulated system.

Contrary to the other existing approaches, UbiREAL [51] provides an emulation framework that allows to combine simulated and real entities. It also provides a 3D graphical renderer to simulate pervasive computing applications. However users have to manually specialize the simulator for every new application area. In contrast, DiaSim relies on DiaSpec to automatically customize the simulation tools (*i.e.*, the editor and renderer).

Context Simulators

Some simulators focus on the simulation of context [9, 45, 61]. The Generic Location Event Simulator publishes location information, which can be used by location-based applications [61]. However, it is limited to location information. SimuContext [9] and Siafu [45] are two other context simulators that go one step further, enabling to define any context types. Siafu also graphically renders simulated environments. However, as a context simulator, Siafu does not provide any support to simulate entities and applications.

Networked Entities Simulators

Various approaches propose to simulate sensor networks [42, 56, 67, 71] and could complement our approach. These simulators provide a more comprehensive support for the simulation of sensors compared to previous approaches. However, they do not consider issues of application development and testing. Network emulators that focus on network-related issues have been proposed [20, 52] and could also complement our approach.

Physically-Accurate Simulation

Existing tools provide partial means to achieve physically-accurate simulation.

Several projects focus on modeling and simulating pervasive computing devices using MATLAB/Simulink [60]. These projects allow a fine-grained modeling of these devices. They provide libraries of digital components that the tester can use to model his

devices. However, they do not allow an analytically sound simulation of the physical environment that impacts such devices. Ptolemy [24] goes one step further than MATLAB/Simulink. Ptolemy provides a library of computation models that the user can compose for modeling and simulating embedded systems. Its main contribution is to allow the simulation of systems that contain heterogeneous computation models. Using Ptolemy would be complementary with our approach. Ptolemy could be used for defining the computation models of devices, whereas DiaSpec describes the outside interface of these devices. However, the support provided by Ptolemy for modeling continuous systems is too restricted for modeling the physical environment of a building.

Other projects focus on modeling the physical environment. COMSOL [19] allows to accurately simulate the surrounding physical environment. For instance, it provides a heat transfer module and an acoustics one. However, these simulations are based on the Finite Element Method and are too slow for modeling the physical environment of a whole building. Other tools allow a faster simulation of the physical properties of a building. Modelica [70] is one of these tools, and is in fact a closely related language to Acumen. The differences between the two are primarily in Acumen's support for binding time separation and partial derivatives [76], but these are in fact orthogonal to the models used here. For more sophisticated models of heat transfer, however, partial derivatives are needed.

Our study of the other simulation approaches showed that testing pervasive computing applications in a simulated physical environment with existing tools is a very complex task. Our approach is a step towards decreasing this complexity by using DSLs to model and execute a pervasive computing application and its surrounding physical environment. Indeed, these DSLs provide the tester with a declarative and high-level support that simplifies the testing of these applications.

11

CONCLUSION

We have presented a tool-based methodology for developing and testing real-size pervasive computing applications. Our methodology provides support throughout the development lifecycle of a pervasive computing application: design, implementation, simulation, and execution. First, the taxonomy of the target area and the architecture descriptions are written in the DiaSpec language. Then, the DiaGen compiler processes these descriptions and generates a dedicated programming framework. This framework raises the abstraction level by providing the programmer with high-level operations for entity discovery and component interactions.

For the testing stage, we have presented a novel approach to simulating pervasive computing applications. We have extended the DiaGen compiler so that it also generates a dedicated simulation programming framework and an emulation layer. The generated emulation layer makes it possible for the same application to be emulated or executed in a real environment. This emulation layer also enables to have an application interact with both real and simulated entities in an hybrid environment. Hybrid simulation allows real entities to be incrementally added in the simulation, as the implementation and deployment progress. The generated simulation programming framework provides support for developing the simulation logic. This logic comprises the simulated entities and the producers of simulated stimuli. A 2D graphical environment is provided to the user to define his simulated environment, simulation scenarios, and to monitor and debug a simulated pervasive computing system. This approach has been implemented in the DiaSim tool, and validated on a large-scale simulation of an engineering school building. We used Acumen's simulation capability to simulate accurately the physical environment. This allowed us to achieve a physically-accurate simulation of pervasive computing applications. We have evaluated DiaSim with respect to its scalability, usability and performance.

Our methodology has been successfully applied to the development of realistic pervasive computing applications in a wide spectrum of areas. Finally, we have generalized our development and testing methodology to a different application domain: the avionics domain.

11.1 ASSESSMENTS

We now assess our tool-based methodology with respect to our initial objectives.

COVERING THE APPLICATION DEVELOPMENT CYCLE Our methodology is based on tools to support the design, implementation and testing stages of the application development cycle. The entities composing a pervasive computing area and application architectures are described using the DiaSpec design language. From this DiaSpec description, the DiaGen code generator then generates a dedicated programming framework to support the implementation stage. Finally, the DiaSim simulator allows to test the application in a simulated physical environment.

ABSTRACTING OVER HETEROGENEITY Our taxonomical approach has been successful at taming the heterogeneity of devices and software components. This is demonstrated by the spectrum of entities modeled to cover the areas of our case study and the ease at implementing entities from their declarations. This approach also showed to be effective for reusing entity declarations across areas.

LEVERAGING AREA-SPECIFIC KNOWLEDGE We provide the area expert with the DiaSpec language to describe the knowledge of a pervasive computing area in the form of a DiaSpec taxonomy. This area description is then leveraged by all the remaining stages of the development cycle. The application architect uses the entities of the taxonomy as building blocks for his application architecture. The developer is provided with a dedicated programming framework to easily implement these entities. Finally, our simulation approach for testing pervasive computing applications is parameterized by this DiaSpec taxonomy.

TRANSPARENT TESTING Our simulation approach makes it possible for the same application code to be simulated or executed in the real environment. We ensure a functional correspondence between a simulated environment and a real one by requiring both implementations to be in conformance with the DiaSpec description of the pervasive computing area. Another benefit of our simulation approach is that it allows the application to be tested in hybrid environments, combining simulated and real entities.

TESTING A WIDE RANGE OF SCENARIOS DiaSim provides a graphical editor to define simulation scenarios for testing pervasive computing applications. Moreover, DiaGen also generates a simulation programming framework. This support provides a default implementation of simulated entities and stimulus producers. However, the tester may use this generated support to implement specific simulated entities and stimulus producers. These specific simulated entities and stimulus producers can then be used in simulation scenarios for testing a pervasive computing application.

SIMULATION RENDERER DiaSim provides a 2D graphical simulation renderer that enables the developer to visually monitor and debug a pervasive computing application.

11.2 ONGOING AND FUTURE WORK

The works that we have presented are being expanded in various directions.

Validating our Simulation Support

To validate our physically-accurate simulation support, we plan on comparing it with the EnergyPlus simulator. EnergyPlus is a simulator of building energy consumption, which has been thoroughly tested and validated. Our goal is to show that, for a given pervasive computing application, the simulation results of the combination of DiaSim and Acumen are the same as EnergyPlus. This would demonstrate that the virtual experiments realized using DiaSim and Acumen are valid.

Simulating Non-Functional Properties

Recent works have extended the DiaSpec language and its compiler to handle non-functional properties. These extensions include (1) handling access conflicts to resources of a pervasive computing system [37], (2) modeling entity failures at the declaration level, enforcing their treatment at the programming level [49], and (3) declaring performance constraints, ensuring them at compile time and run time [32]. We plan to extend our simulation support to allow the simulation of these non-functional properties. For instance, we would like to allow the tester to create stimulus producers of entity failures to test how the application reacts in case of entity failures.

Adding Human Behavior Modeling

Numerous pervasive computing applications surround people in their life. To help simulating this kind of applications, we plan on connecting human behavior models to DiaSim. For instance, human behaviors could be implemented in a multi-agent simulation toolkit such as MASON [43]. Each agent would represent a simulated person. Coupling DiaSim with human behavior would allow the simulation of an evacuation plan in case of a life-threatening situation. For example, the simulation of the building occupants would enable to observe their behaviors in case of fire.

Enhancing the system monitoring

A pervasive computing system may involve a large number of entities and applications. Monitoring such a system rapidly becomes excessively complicated. In particular, large-scale simulations in which numerous events occur at the same time are hard to monitor, even with graphical rendering and logs. To enhance monitoring, we would like to add contracts to DiaSpec in the form of pre- and post-conditions to entities, controllers, and contexts. These contracts would drive the rendering of a simulation by drawing the tester's attention when they are violated.

Enhancing the graphical renderer

The 2D graphical rendering provided by DiaSim allows to easily monitor the simulated applications. However, the user experience of these simulated applications would be improved with a 3D graphical rendering. Indeed, users would be able to test applications immersed in a simulated 3D physical environment. We plan to render simulations in 3D using Blender [6], an authoring tool for creating 3D animations and video games.

Enhancing the Testing Support

We plan to simplify the testing phase by automatically generating a dedicated unit testing framework with mock objects [44] from the DiaSpec description. In accordance with the architect, a tester could then describe the desired behavior of each entity and component separately, even before the implementation has started. Each developer would then be able to assess the correctness of their implementations by running the tests. It would

nicely complement our current testing support, as it only allows to test the complete application and not each component separately.

Verification

Another promising direction is to take advantage of architectural invariants for guiding program analysis tools. Our generative approach could automatically add architectural invariants as axioms to the model, facilitating verification. For example, we are investigating the verification of safety properties by injecting the architectural invariants from the DiaSpec specification in the model checker JPF [75].

Empirical evaluation

We have showed that our tool-based methodology can be used in a wide range of pervasive computing areas. This gives hints of the usability of our methodology. We plan to conduct an empirical evaluation based on a well-defined experimental methodology. In particular, we would like to evaluate the usability and productivity gained by comparing our approach with existing tool-based development methodologies for pervasive computing applications.

BIBLIOGRAPHY

[1] Jonathan Aldrich, Craig Chambers, et David Notkin. ArchJava: Connecting software architecture to implementation. Dans *ICSE'02: Proceedings of the 24th International Conference on Software Engineering*, pages 187–197, New York, NY, USA, 2002.

[2] ANTLR website. Another tool for language recognition, 2012. url: `http://www.antlr.org/`.

[3] Apple. iOS SDK, 2012. `https://developer.apple.com/technologies/ios/`.

[4] J. J. Barton et V. Vijayaraghavan. Ubiwise, a ubiquitous wireless infrastructure simulation environment. Technical report, Hewlett Packard, 2002.

[5] Benjamin Bertran, Charles Consel, Wilfried Jouve, Hongyu Guan, et Patrice Kadionik. SIP as a universal communication bus: A methodology and an experimental study. Dans *ICC'10: Proceedings of the 9th International Conference on Communications*, Cape Town, South Africa, 2010.

[6] Blender. Blender website, 2012. url: `http://www.blender.org`.

[7] W. Bobenhausen. *Simplified design of HVAC systems*. John Wiley and Sons, 1994.

[8] Grady Booch, James Rumbaugh, et Ivar Jacobson. *The Unified Modeling Language User Guide (2nd Edition)*. Addison Wesley, 2005.

[9] T. Broens et A. van Halteren. SimuContext: Simply Simulate Context. Dans *ICAS '06: Proceedings of the International Conference on Autonomic and Autonomous Systems*, page 45, 2006.

[10] Yuriy Brun, Giovanna Marzo Serugendo, Cristina Gacek, Holger Giese, Holger Kienle, Marin Litoiu, Hausi Müller, Mauro Pezzè, et Mary Shaw. Engineering Self-Adaptive Systems through Feedback Loops. Dans Betty H. Cheng, Rogério Lemos, Holger Giese, Paola Inverardi, et Jeff Magee, editors, *Software Engineering for Self-Adaptive Systems*, pages 48–70. Springer-Verlag, Berlin, Heidelberg, 2009. isbn: 978-3-642-02160-2. doi: http://dx.doi.org/10.1007/978-3-642-02161-9_3.

[11] Julien Bruneau, Wilfried Jouve, et Charles Consel. DiaSim, a parameterized simulator for pervasive computing applications. Dans *Mobiquitous'09: Proceedings of the 6th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, pages 1–10. 2009.

[12] Julien Bruneau, Charles Consel, Marcia O'Malley, Walid Taha, et Wail Masry Hannourah. Preliminary results in virtual testing for smart buildings (poster). Dans *Mobiquitous'10: Proceedings of the 7th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, 2010.

[13] Julien Bruneau, Quentin Enard, Stéphanie Gatti, Emilie Balland, et Charles Consel. Design-driven Development of Safety-critical Applications: A Case Study In Avionics. Technical report, Phoenix Research Group, INRIA Bordeaux, France, 2011.

[14] Damien Cassou, Benjamin Bertran, Nicolas Loriant, et Charles Consel. A generative programming approach to developing pervasive computing systems. Dans *GPCE'09: Proceedings of the 8th International Conference on Generative Programming and Component Engineering*, pages 137–146, Denver, CO, USA, 2009.

[15] Damien Cassou, Emilie Balland, Charles Consel, et Julia Lawall. Leveraging architectures to guide and verify development of sense/compute/control applications. Dans *ICSE'11: Proceedings of the 33rd International Conference on Software Engineering*. 2011. to appear.

[16] Damien Cassou, Julien Bruneau, Charles Consel, et Emilie Balland. Towards a Tool-based Development Methodology for Pervasive Computing Applications. *IEEE Transactions on Software Engineering*, 2011.

[17] Guanling Chen et David Kotz. Context aggregation and dissemination in ubiquitous computing systems. Dans *WMCSA'02: Proceedings of the 4th Workshop on Mobile Computing Systems and Applications*, pages 105–114, Washington, DC, USA, 2002.

[18] Toby H.J. Collett, Bruce A. MacDonald, et Brian P. Gerkey. Player 2.0: Toward a practical robot programming framework. Dans *ACRA'05: Proceedings of the 7th Australasian Conference on Robotics and Automation*, pages 1–9, Sydney, Australia, 2005.

[19] COMSOL. COMSOL: Multiphysics Modeling, Finte Element Analysis, and Engineering Simulation Software, 2012. http://www.comsol.com/.

[20] F. D'Aprano, M. de Leoni, et M. Mecella. Emulating Mobile Ad-hoc Networks of Hand-Held Devices: the OCTOPUS Virtual Environment. Dans *MobiEval'07: Proceedings of the 1st International Workshop on System Evaluation for Mobile Platforms*, pages 35–40, 2007.

[21] Anind K. Dey, Gregory D. Abowd, et Daniel Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction*, 16(2):97–166, 2001.

[22] Z. Drey et C. Consel. A visual, open-ended approach to prototyping ubiquitous computing applications. Dans *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2010 8th IEEE International Conference on*, 2010. doi: 10.1109/PERCOMW.2010.5470549.

[23] Zoé Drey, Julien Mercadal, et Charles Consel. A taxonomy-driven approach to visually prototyping pervasive computing applications. Dans *DSL WC'09: Proceedings of the 1st Working Conference on Domain-Specific Languages*, pages 78–99, 2009.

[24] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, et Y. Xiong. Taming heterogeneity - The Ptolemy Approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[25] EnergyStar. A Guide to Energy-Efficient Heating and Cooling by Energy Star, 2009. http://www.energystar.gov/index.cfm?c=heat_cool.pr_hvac.

[26] EnterpriseArchitect. Enterprise Architect - UML design tools and UML CASE tools for software development, 2012. url: http://www.sparxsystems.com.au/products/ea/index.html.

[27] Wireshark Foundation. Wireshark: A Network Protocol Analyzer, 2012. http://www.wireshark.org.

[28] Martin Fowler. UML mode, 2003. url: http://www.martinfowler.com/bliki/UmlMode.html.

[29] Martin Fowler. Fluent interface, 2005. url: http://www.martinfowler.com/bliki/FluentInterface.html.

[30] R. E. Frechette et R. Gilchrist. Towards zero energy, a case study: Pearl River Tower, Guangzhou, China. Dans *CTBUH:*

Proceedings of the Council on Tall Buildings and Urban Habitat's 8th World Congress, pages 7–16, 2008.

[31] Erich Gamma, Richard Helm, Ralph Johnson, et John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[32] Stéphanie Gatti, Emilie Balland, et Charles Consel. A stepwise approach for integrating QoS throughout software development. Dans *FASE'11: Proceedings of the 14th European Conference on Fundamental Approaches to Software Engineering*, 2011.

[33] Google. Android SDK, 2012. `http://developer.android.com`.

[34] R. Grimm. One.world: Experiences with a Pervasive Computing Architecture. *IEEE Pervasive Computing*, 3(3):22–30, 2004.

[35] Karen Henricksen et Jadwiga Indulska. A software engineering framework for context-aware pervasive computing. Dans *PERCOM'04: Proceedings of the 2nd International Conference on Pervasive Computing and Communications*, pages 77–86. 2004.

[36] Jim Highsmith et Martin Fowler. The agile manifesto. *Software Development Magazine*, 9(8):29–30, 2001.

[37] Henner Jakob, Charles Consel, et Nicolas Loriant. Architecturing Conflict Handling of Pervasive Computing Resources. Dans *DAIS'11: 11th IFIP International Conference on Distributed Applications and Interoperable Systems*, 2011.

[38] W. Jouve, J. Bruneau, et C. Consel. DiaSim: A Parameterized Simulator for Pervasive Computing Applications (Demo). Dans *PERCOM'09: Proceedings of the 7th IEEE International Conference on Pervasive Computing and Communications*, 2009.

[39] Jevgeni Kabanov et Rein Raudjärv. Embedded typesafe domain specific languages for Java. Dans *PPPJ'08: Proceedings of the 6th International Symposium on Principles and Practice of Programming in Java*, pages 189–197, New York, NY, USA, 2008.

[40] Matthias Kranz, Radu Bogdan Rusu, Alexis Maldonado, Michael Beetz, et Albrecht Schmidt. A player/stage system for context-aware intelligent environments. Dans *UbiSys'06: Proceedings of the System Support for Ubiquitous Computing Workshop*, pages 1–7, 2006.

[41] T. Kuehn. *Fundamentals: 2005 Ashrae Handbook*. Amer Society of Heating, 2005.

[42] P. Levis, N. Lee, M. Welsh, et D. Culler. TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. Dans *SenSys '03: Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, pages 126–137, 2003.

[43] S. Luke, C. Cioffi-Revilla, L. Panait, K. Sullivan, et G. Balan. MASON: A Multiagent Simulation Environment. *Simulation*, 81(7):517, 2005.

[44] Tim Mackinnon, Steve Freeman, et Philip Craig. *Endo-testing: Unit testing with mock objects*, chapter 17, pages 287–301. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[45] Miquel Martin et Petteri Nurmi. A generic large scale simulator for ubiquitous computing (poster). Dans *MobiQuitous'06: Proceedings of the 3rd International Conference on Mobile and Ubiquitous Systems: Networking & Services*, pages 1–3, San Jose, CA, USA, 2006.

[46] Ted McFadden, Karen Henricksen, Jadwiga Indulska, et Peter Mascaro. Applying a disciplined approach to the development of a context-aware communication application. Dans *PERCOM'05: Proceedings of the 3rd International Conference on Pervasive Computing and Communications*, pages 300–306. 2005.

[47] Nenad Medvidovic et Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1): 70–93, 2000.

[48] Nenad Medvidovic, Eric M. Dashofy, et Richard N. Taylor. Moving architectural description from under the technology lamppost. *Information and Software Technology*, 49:12–31, 2007.

[49] Julien Mercadal, Quentin Enard, Charles Consel, et Nicolas Loriant. A Domain-Specific Approach to Architecturing Error Handling in Pervasive Computing. Dans *OOPSLA'10: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 2010.

[50] Ricardo Morla et Nigel Davies. Evaluating a location-based application: A hybrid test and simulation environment. *IEEE Pervasive Computing*, 3(3):48–56, 2004.

[51] H. Nishikawa, S. Yamamoto, M. Tamai, K. Nishigaki, T. Kitani, N. Shibata, K. Yasumoto, et M. Ito. UbiREAL: Realistic

Smartspace Simulator for Systematic Testing. Dans *UBICOMP'06: Proceedings of the 8th International Conference on Ubiquitous Computing*, pages 459–476, 2006.

[52] NS-2. NS-2 Network Simulator, 2011. url: `http://nsnam.isi.edu/nsnam/index.php/Main_Page`.

[53] Eleanor O'Neill, Martin Klepal, David Lewis, Tony O'Donnell, Declan O'Sullivan, et Dirk Pesch. A testbed for evaluating human interaction with ubiquitous computing environments. Dans *TRIDENTCOM'05: Proceedings of the 1st International Conference on Testbeds and Research Infrastructures for the DEvelopment of NeTworks and COMmunities*, pages 60–69. 2005.

[54] Alexander R. Perry. The FlightGear Flight Simulator. Dans *Proceedings of the USENIX Annual Technical Conference*, 2004.

[55] Ruben Picek et Vjeran Strahonja. Model Driven Development - future or failure of software development? Dans *IIS'07: Proceedings of the 18th International Conference on Information and Intelligent Systems*, pages 407–413, Varazdin, Croatia, 2007.

[56] J. Polley, D. Blazakis, J. McGee, D. Rusk, J. S. Baras, , et M. Karir. ATEMU: A Fine-Grained Sensor Network Simulator. Dans *SECON'04: Proceedings of the 1st IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks*, 2004.

[57] Anand Ranganathan et Roy H. Campbell. Advertising in a pervasive computing environment. Dans *WMC'02: Proceedings of the 2nd International workshop on Mobile commerce*, pages 10–14. 2002.

[58] Anand Ranganathan, Shiva Chetan, Jalal Al-Muhtadi, Roy H. Campbell, et M. Dennis Mickunas. Olympus: A high-level programming model for pervasive computing environments. Dans *PERCOM'05: Proceedings of the 3rd International Conference on Pervasive Computing and Communications*, pages 7–16. 2005.

[59] Vinny Reynolds, Vinny Cahill, et Aline Senart. Requirements for an ubiquitous computing simulation and emulation environment. Dans *InterSense'06: Proceedings of the 1st International Conference on Integrated Internet Ad hoc and Sensor Networks*, New York, NY, USA, 2006. doi: 10.1145/1142680.1142682.

[60] P. Riederer. MATLAB/Simulink for Building and HVAC Simulation - State of the Art. Dans *Proceedings of the 9th International IBPSA Conferene*, 2005.

[61] K. Sanmugalingam et G. Coulouris. A Generic Location Event Simulator. Dans *UBICOMP'02: Proceedings of the 4th International Conference on Ubiquitous Computing*, pages 308–315, 2002.

[62] Albrecht Schmidt, Kofi Asante Aidoo, Antti Takaluoma, Urpo Tuomela, Kristof Van Laerhoven, et Walter Van de Velde. Advanced interaction in context. Dans *HUC'99: Proceedings of the 1st International Symposium on Handheld and Ubiquitous Computing*, pages 89–101, London, UK, 1999.

[63] Douglas C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31, 2006. doi: 10.1109/MC.2006.58.

[64] Estefanía Serral, Pedro Valderas, et Vicente Pelechano. Towards the model driven development of context-aware pervasive systems. *Pervasive and Mobile Computing*, 6:254–280, 2010. doi: 10.1016/j.pmcj.2009.07.006.

[65] Mary Shaw. Beyond Objects: A Software Design Paradigm Based on Process Control. *SIGSOFT Software Engineering Notes*, 20:27–38, 1995.

[66] StringTemplate website. Stringtemplate template engine, 2012. url: `http://www.stringtemplate.org/`.

[67] Sameer Sundresh, Wooyoung Kim, et Gul Agha. Sens: A sensor, environment and network simulator. Dans *Proceedings of the 37th Annual Simulation Symposium*, pages 221–230, Los Alamitos, CA, USA, 2004.

[68] R. N. Taylor, N. Medvidovic, et E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2009.

[69] Dave Thomas. MDA: Revenge of the modelers or UML utopia? *IEEE Software*, 21:15–17, 2004. doi: 10.1109/MS.2004.1293067.

[70] M. Tiller. *Introduction to Physical Modeling With Modelica*. Kluwer Academic Publishers, 2001.

[71] B. L. Titzer, D. K. Lee, et J. Palsberg. Avrora: Scalable Sensor Network Simulation With Precise Timing. Dans *IPSN'05: Proceedings of the 4th International Symposium on Information Processing in Sensor Networks*, pages 477–482, 2005.

[72] Naoyasu Ubayashi, Jun Nomura, et Tetsuo Tamai. Archface: A contract place where architectural design and code meet together. Dans *ICSE'10: Proceedings of the 32nd International Conference on Software Engineering*, pages 75–84, New York, NY, USA, 2010.

[73] U.S. Departement of Energy. EnergyPlus Energy Simulation Software, 2011. url: http://apps1.eere.energy.gov/buildings/energyplus.

[74] Eric. Van Wyk, Lijesh Krishnan, Derek Bodin, et August Schwerdfeger. Attribute grammar-based language extensions for java. Dans *ECOOP'07: Proceedings of the 21th European Conference on Object-Oriented Programming*, page 575. 2007.

[75] Willem Visser, Klaus Havelund, Guillaume Brat, et SeungJoon Park. Model checking programs. Dans *ASE'00: Proceedings of the 15th International Conference on Automated Software Engineering*, pages 3–12, Washington, DC, USA, 2000.

[76] Y. Zhu, E. Westbrook, J. Inoue, A. Chapoutot, C. Salama, M. Peralta, T. Martin, W. Taha, M. O'Malley, R. Cartwright, A. Ames, et R. Bhattacharya. Mathematical equations as executable models of mechnical systems. Dans *ICCPS'10: Proceedings of the 1st International Conference on Cyber-Physical Systems*, 2010.